**History of React**

**What is react?**

React is a JavaScript library

**ES6- class** declarations in 2015, then JavaScript support class was release. After release this class in javascript, react developer uses class into react. before they use React.CreateClass()… after ES-6 module, they uses **class** keyword and extends React.Component. we need to call super class inside class component.

**Before there was many problems:** complex states, lifecycle methods, sharing same logic, duplicate code

Constructor(props){ super(props)}.

**Fundamental concepts:** States, props, components, compositions,

**React hook**s introduced in 2019 Feb 16. V16.8

useState() hooks gives us ability to manage states. And useEffect() gives us to manage the side effect, side effect means which is not related to User Interface. React mainly developed to build UI. So react give us useEffect() hooks to manage the side effect that is not part of the UI creation.

**useEffect(callback, dependencies)** takes two parameter. First time useEffect() run one time. But when any dependencies is changed the useEffect() is called.

**Now we have solution of many problems:**

States management using **useState()** hooks, instead write state variable inside class component. Lifecycle methods using **useEffect(),** instead writing same logic code like didMount… disMountUpdate.. etc for side effect or data fetching etc. duplicate codes reduce by creating custom hooks, instead writing same code for many times for many data fetching. sharing same logic through hooks using context api, instead writing a class or props rendering or higher order class component.

**Some points to note:**

* Hooks are available from react version 16.8+
* Hooks don’t contain any breaking changes & it’s 100% backward compatible.
* We can not write hooks inside class components
* Hooks don’t replace your existing knowledge. It just provides a more direct API to the React concepts you already know.

**useState() hooks**

it return an array of two element. First element is state, and second element is a function. We need to destructure it like const [state, setState] = useState(initialState value).

|  |
| --- |
| import React, { useState } from "react";  const Todo = () => {    const [todo, setTodo] = useState("");    const [warning, setWarning] = useState("");    const handleInput = (e) => {      const inputValue = e.target.value;      const updateWarning = inputValue.includes(".j")        ? "Your need JavaScript skill to complete the task. Do you have it?"        : null;      // when setTodo() is called, then rerender the component but react skipp above state, it cannot re assign again. react memorize the state value. react render the component from the return method      setTodo(inputValue);      setWarning(updateWarning);    };    return (      <div>        <p>{todo}</p>        <p>          <textarea            onChange={handleInput}            name="todo"            value={todo}            cols="30"            rows="10"          ></textarea>        </p>        <hr />        {/\* if there any warning then it show. else show 'good choice' \*/}        <h2>{warning || "Good choice"}</h2>      </div>    );  };  export default Todo; |

**Some rules of hooks:**

1. Do not call **useState()** hooks inside any function, or inside if condition.
2. Always use hook in top level.
3. Use hook in react area.
4. In class component, state would merge. But state value is not merge. So change in state must return full state value. Before + new .
5. const [todo, setTodo] = useState({
6. title: "todo app",
7. description: "this is todo app description",
8. });

setTodo(..todo, description: 'this is new description');

when we want to change any inside todo state. Must provide previous and update both inside setState() method. setState({…todo, titile:’this is new title’});

**use previous state value:**

  const [count, setCount] = useState(10);

we need to use a callback function inside **setCount((prev) => prev+1);** this is the right approach.

 setCount((prev) => prev + 1);

**useEffect()** side effect hooks

react responsibilities is UI render, React is UI library. Secondly based in user input it react. or reactive.

It generate JSX element, manage states for UI.

Also it done side effect related work using useEffect(), like data fetch from any api, updating DOM, setting any subscription, timers set,

Before useEffect(), we need to use componentDidMount(), componentDidUpdate(), componentWillUnmount() etc method need to use. These are lifecycle method

But here is problem is that, we need to write duplicate code, repeating code etc.

**Solution:** useEffect() solved above problem. Organized, unique code, perform side effect, replaces componentDidMount(), componentDidUpdate(), componentWillUnmount()

**How useEffect works?**

useEffect() is a simple function. It takes two parameter, first parameter is function and second parameter is dependencies array. When any changes occur then useEffect() re render it is inner function. When dependencies is changed, useEffect() called it is first parameter function.

    useEffect(() => {

        const response = fetchData(api);

        setTodo(...prevTodo, response);

    }, [todo])

We pass only function , dependencies is optional.

**Note: blank array**: it only call first time. componentDidMount()

**Array with dependencies:** it re render when dependencies value is changed. **ComponentWillUpdate()**

**No array:** it always re render.

**Note:** when a component will unmount, we need to handle useEffect() render. Is class component, componentWillMount() will clear the interval that was set inside componentDidMount() method.

But in useEffect(), we need to cleanup the useEffect first parameter, this function need to return. Return an anonymous function. It works similar to the componentWillUnmount() method. Mainly when component unmount, the first parameter ( function) will return a function. Mainly this function return function when component is unmount. Inside this function we can do many thing that is need to do something to avoid re render even component is unmount.

**useCallback() and useMemo()**

useMemo() and useCallback() help us to optimized performance. It reduce unnecessary re render.

**Note:** useMemo() is not a hook. This is simple higher order function. We can solved to stop re-render.

last portion of the component we need to use React.memo() function and need to pass this component which component want to stop re-render.

We need to export default **React.memo(component)**

function App() {

  console.log("app rendering");

  const [count0, setCount0] = useState(0);

  const [count1, setCount1] = useState(0);

  const incrementByOne = () => {

    setCount0((prev) => prev + 1);

  };

  const incrementByFive = () => {

    setCount1((prev) => prev + 5);

  };

  return (

    <div className="App">

      <header className="App-header">

        <Title />

        {/\* <Todo /> \*/}

        <Count count={count0} />

        <Button title={"Counter 1"} handleClick={incrementByOne} />

        <Count count={count1} />

        <Button title={"Counter 2"} handleClick={incrementByFive} />

      </header>

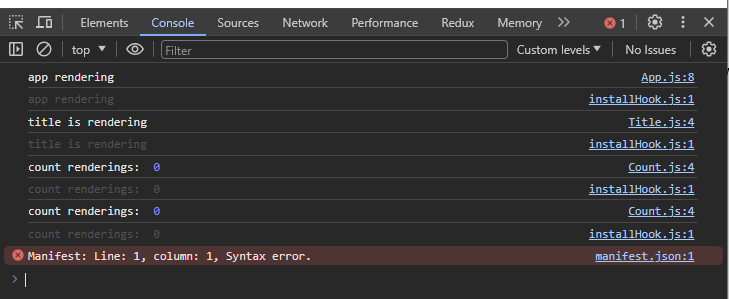
    </div>

  );

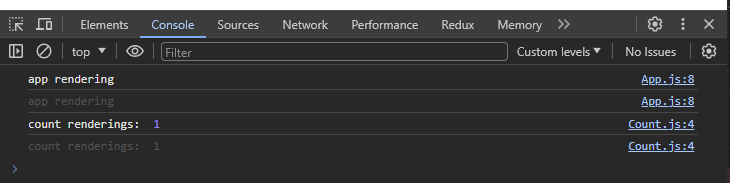
}

In Title, Count, and button I have used memo().. now it will not re render when any changed in like Count. It just changed only specific Count.

Before:



After:



**useCallback() it take two parameter**, one is a callback function and other is dependencies array. When dependedcies array is changed, then react forgot the reference of this **incrementByOne** function.

  const incrementByOne = useCallback(() => {

    setCount0((prev) => prev + 1);

  }, []);

**useMemo()** is used to memorize some result that is not necessary when component is re render for another task. Means I have two event. When event one will perform. One huge calculation will be done. Means only this calculation will done only based on even one. But when even two will perform. Component will re render, but the calculation that is already calculated for even one. But when even two is perform. The calculation again perform calculation due to re render nature. But even two only perform another task. No need to again calculate calculation part that is mainly related to the even one. We can use useMemo() to memorize a part of calculation that is mandatory for other even. When component will re render, the memorize part only use the previous calculated part. it will not again calculate. We can pass dependencies that is related to even one. When only even one is perform. Only forgot previous memorized value. Again calculate memorized value.

Example:

const isEvenOdd = () => {

    let i = 0;

    while (i < 1111122211) i++;

    return count0 % 2 === 0;

  };

This function is mainly related to count0 state. this calculation done based on only count0.

return (

    <div className="App">

      <header className="App-header">

        <Title />

        {/\* <Todo /> \*/}

        <span>{isEvenOdd() ? "even" : "odd"}</span>

        <Count count={count0} />

        <Button title={"Counter 1"} handleClick={incrementByOne} />

        <Count count={count1} />

        <Button title={"Counter 2"} handleClick={incrementByFive} />

      </header>

    </div>

  );

First time isEvenOdd() is called. When count is changed, the app component will re render, and isEvenOdd again calculate. When counter two is changed again app component will re render and isEvenOdd() again called. And again it run many times. Notice here only calculation is different when the count0 is changes. No need to again calculation when count1 is changed. We can memorize the previous value when count1 is changed. useMemo() take two parameter, one is callback, and second is dependencies. When dependencies is changed only forgot previous value. Otherwise useMemo() return only previous memorized value.

Here is the updated code:

// useMemo() to memorize the calculation value for further use when component is rendered again for other task. not event and

  const isEvenOdd = useMemo(() => {

    let i = 0;

    while (i < 1111122211) i++;

    return count0 % 2 === 0;

  }, [count0]);

Now isEvenOdd() only recalculate when dependencies is changed

Otherwise it return the memorized value

**useRef() hook**

we can perform DOM operation using useRef(). Direct we should not perform DOM manipulation in react code. React give us useRef() hook to perform dom operation like set attribute, input focus etc. react way DOM manipulation is good in react code.

**Example Problem:** we have a form, first input field must be focus when first page is loading.

Note: mainly we catch this input field when page is loaded done. In class component we can done easily using componentDidMount() method. Since this is our functional component we need to use useEffect() hook. It done similar job. When component is mounted this useEffect() will be fired. So we must need to use useRef() insdie the useEffect() hook.

const Forms = () => {

  // initially create a simple ref variable, it is just a variable, when component render is done, where this variable is used, those element DOM node will connect into this inputRef's current property.

  const inputRef = useRef(null);

  useEffect(() => {

    // component did load successfully

    // do the DOM manipulation using useRef()

    console.log(inputRef.current.focus());

    // we can now do the DOM manipulation, that we could use in raw javascript DOM manipulation

  }, []);

  return (

    <div>

      <input ref={inputRef} type="text" placeholder="Enter something" />

      <input type="text" placeholder="another input field" />

    </div>

  );

};

**Note:** we can no pass the useRef() variable into another component or other input field that is outside fo this component. We need to use

Const forwardInput = React.forwardRef(component); this HOC(higher order component) return a function. We need to export this function.

Function inputField = ({type, placeholder}, ref) {……… component };

Instead export inputField. Need to export forwardInput function. Another this is that, we need to pass ref into inputField component.

**Another use case** of useRef().

React can push value into ref.current also we can push data into ref variable. Even component is rerender but ref variable is not changeable. We can keep data that is mutable, even state is changed cause re render of a component.

If we want to use useRef variable, do not use this ref variable to reference any react element.

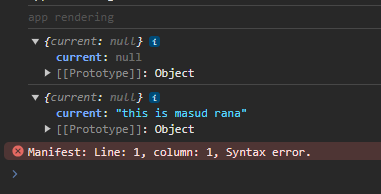
  const inputRef = useRef(null);

  console.log(inputRef);

  useEffect(() => {

    inputRef.current = "this is masud rana"; }

even the ref variable is changing the component will not re render, ref variable have not any interfere to component re-rendering. Even state is re-rendered the ref variable always keep the previous value. This is one kind of storage.



Here is the useRef output when I have consoled it to see the result.

**useReducer() hook**

useReducer() is used to manage state. it is a React hook. useReducer() come before useState(). useState() is created based on useReducer() hook. useState() is simple version is useReducer()

**what is the difference and when which should I use?**

useReducer() mainly used for reducer. In JavaScript we have reducer() function. Array.prototype.reducer(). This is built in method.

useReducer(reducer, initialState). And return two variable. One is newState, and dispatch() function.

Reducer(state, action){… mainly switch case}…. Based on action, it return new state.

Here is the example: it is like redux state management

import React, { useReducer } from "react";

// initialState

const initialState = 0;

// reducer

const reducer = (state, action) => {

  switch (action.type) {

    case "increment":

      return state + 1;

    case "decrement":

      return state - 1;

    default:

      return state;

  }

};

const Counter = () => {

    // useReducer

  const [newState, dispatch] = useReducer(reducer, initialState);

  return (

    <div>

      <h3>{newState}</h3>

       {/\* dispatch action \*/}

      <button onClick={() => dispatch({ type: "increment" })}>

        Increment{" "}

      </button>

      <button onClick={() => dispatch({ type: "decrement" })}>

        decrement{" "}

      </button>

    </div>

  );

};

export default Counter;

**here is the another example of useReducer() hook with intialState as an object:**

import React, { useReducer } from "react";

// initialState1

const initialState = {

  counter1: 0,

  counter2: 0,

};

// reducer

const reducer = (state, action) => {

  switch (action.type) {

    case "increment1": {

      return {

        ...state,

        counter1: state.counter1 + action.value,

      };

    }

    case "decrement1": {

      return {

        ...state,

        counter1: state.counter1 - action.value,

      };

    }

    case "increment2": {

      return {

        ...state,

        counter2: state.counter2 + action.value,

      };

    }

    case "decrement2": {

      return {

        ...state,

        counter2: state.counter2 - action.value,

      };

    }

    default:

      break;

  }

};

const ComplexCounter = () => {

  const [newState, dispatch] = useReducer(reducer, initialState);

  return (

    <div>

      <h3>1st Counter: {newState.counter1}</h3>

      <button onClick={() => dispatch({ type: "increment1", value: 1 })}>

        Increment by 1

      </button>

      <button onClick={() => dispatch({ type: "decrement1", value: 5 })}>

        decrement by 5

      </button>

      <h3>2nd Counter: {newState.counter2}</h3>

      <button onClick={() => dispatch({ type: "increment2", value: 5 })}>

        Increment by 5

      </button>

      <button onClick={() => dispatch({ type: "decrement2", value: 2 })}>

        decrement by 2

      </button>

    </div>

  );

};

export default ComplexCounter;

**Now we will see another way to manage multiple state management using useReducer() hook**

import React, { useReducer } from "react";

// initialState

const counter1 = 0;

const counter2 = 5;

// reducer

const reducer = (state, action) => {

  switch (action.type) {

    case "increment":

      return state + 1;

    case "decrement":

      return state - 1;

    default:

      return state;

  }

};

const CounterThree = () => {

  // useReducer

  const [newState, dispatch] = useReducer(reducer, counter1);

  // we can use similar reducer with different state variable passing into useReducer()  and get different new state and dispatch

  const [newState2, dispatch2] = useReducer(reducer, counter2);

  return (

    <div>

      <h3>{newState}</h3>

      {/\* dispatch action \*/}

      <button onClick={() => dispatch({ type: "increment" })}>

        Increment{" "}

      </button>

      <h3>new counter start with 5: {newState2}</h3>

      <button onClick={() => dispatch2({ type: "increment" })}>

        Increment{" "}

      </button>

    </div>

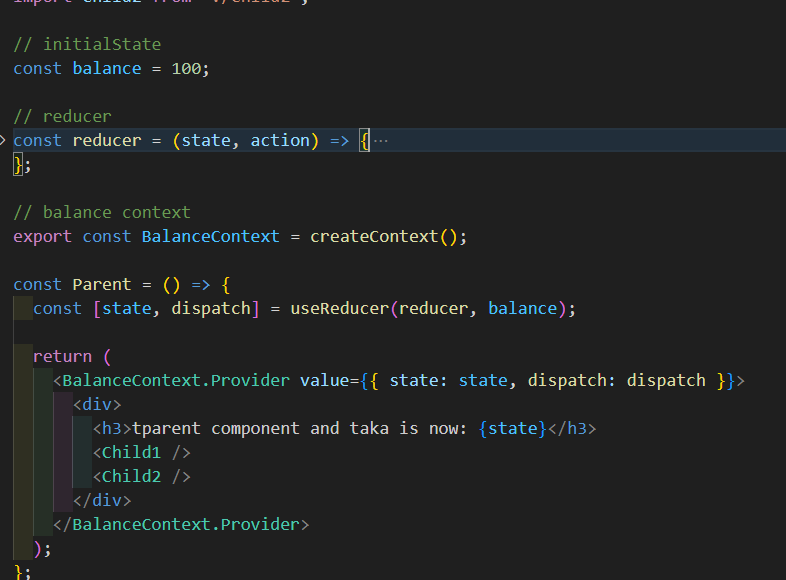
  );

};

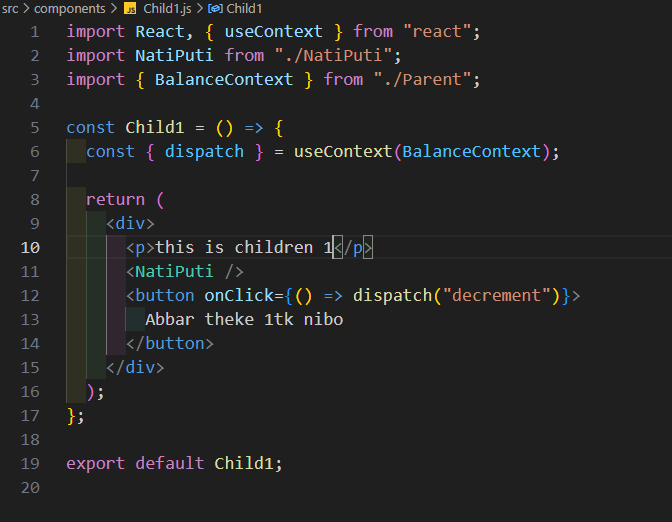
export default CounterThree;

**useReducer() in global state management**

**wrap parent component with context and pass the state and dispatch function**



**Use context value from the parent and can edit using useContext() hook.**



**Which case we need to use useState() and useReducer()?**

1. When states change many, use useReducer instead useState()
2. If states value is primitives like number, string, Boolean
3. If we have complex logic, use useReducer()
4. if state scope is local, use useState()
5. if state scope is global, useReducer()

**Custom hook**

To build a custom hook. Start with use. React itself suggest it. useWindowWidth() is a custom hook.

Since this is custom hook. We can return anything Js code. No need jsx. Since this is custom hooks.

import { useEffect, useState } from "react";

const useWindowWidth = (size) => {

  const [smallScreenSize, setSmallScreenSize] = useState(false);

  // or we can use useCallback

  useEffect(() => {

    const checkScreenSize = () => {

      setSmallScreenSize(window.innerWidth < size);

    };

    checkScreenSize();

    window.addEventListener("resize", checkScreenSize);

  }, [size]);

  return smallScreenSize;

};

export default useWindowWidth;