# use of the map method

import React from 'react';

function App() {

  const items = [

    { id: 1, name: "Apple" },

    { id: 2, name: "Banana" },

    { id: 3, name: "Cherry" }

  ];

  return (

    <ul>

      {items.map(item => (

        <li key={item.id}>{item.name}</li>

      ))}

    </ul>

  );

}

export default App;

# Passing Data via Props Props in React

import React from 'react';

// Child component

function Header({ title }) {

  return <h1>{title}</h1>;

}

// Parent component

function App() {

  return (

    <div>

      <Header title="My App" />

    </div>

  );

}

export default App;

# Passing Components as Children

import React from 'react';

// Container component

function Container(props) {

  return (

    <div className="container-style">

      {props.children}  {/\* Accessing children prop \*/}

    </div>

  );

}

// App component

function App() {

  return (

    <Container>

      <h1>Welcome to My App</h1>

      <p>This content is passed as children to the Container component.</p>

    </Container>

  );

}

export default App;

# Handling Events

import React, { useState } from 'react';

function App() {

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

  // Event handler function

  const handleClick = () => {

    setCount(count + 1);

  };

  return (

    <div>

      <h1>Count: {count}</h1>

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

    </div>

  );

}

export default App;

# Passing functions via props

import React, { useState } from 'react';

// Child component

function Child({ onButtonClick }) {

  return (

    <button onClick={onButtonClick}>Click Me</button>

  );

}

// Parent component

function App() {

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

  // Function to handle the button click

  const handleClick = () => {

    setMessage("Button was clicked!");

  };

  return (

    <div>

      <h1>{message}</h1>

      <Child onButtonClick={handleClick} />

    </div>

  );

}

export default App;

# useState

import React, { useState } from 'react';

function Counter() {

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

  return (

    <div>

      <h1>Count: {count}</h1>

      <button onClick={() => setCount(count + 1)}>Increment</button>

    </div>

  );

}

export default Counter;

If the Counter component manages its own state (like in the previous example) and nothing in the homepage component depends on the state of the Counter, then only the Counter will re-render. The homepage itself will not re-render unless its own state or props change.

import React, { useState } from 'react';

function App1() {

  const [counter, setCounter] = useState(15); // Initial counter value set to 15

  const addValue = () => {

    setCounter(counter + 1);

    setCounter(counter + 1);

    setCounter(counter + 1);

    setCounter(counter + 1);

  };

  FIXME: OUTPUT=16

  return (

    <div>

      <p>Counter: {counter}</p>

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

    </div>

  );

}

export default App1;

import React, { useState } from 'react';

function App2() {

  const [counter, setCounter] = useState(15); // Initial value set to 15

  const addValue = () => {

    setCounter(prevCounter => prevCounter + 1);

    setCounter(prevCounter => prevCounter + 1);

    setCounter(prevCounter => prevCounter + 1);

    setCounter(prevCounter => prevCounter + 1);

  };

  FIXME: OUTPUT=19

  return (

    <div>

      <p>Counter: {counter}</p>

      <button onClick={addValue}>Increment by 4</button>

    </div>

  );

}

module.exports = {

    app1,

    app2

}

import React from 'react';

import Counter from './Counter';

function HomePage() {

  return (

    <div>

      <h1>Welcome to My Homepage</h1>

      <Counter /> {/\* This will re-render independently \*/}

    </div>

  );

}

export default HomePage;

If the Counter is managing its state but the homepage has some dependency on that state (for example, if the homepage passes props to Counter based on its state), then a state change in Counter could cause the homepage to re-render as well.

import React, { useState } from 'react';

import Counter from './Counter';

function HomePage() {

  const [message, setMessage] = useState("Welcome!");

  return (

    <div>

      <h1>{message}</h1>

      <Counter onCounterChange={() => setMessage("Counter updated!")} />

    </div>

  );

}

export default HomePage;

# React Form

In React, when you update the state using the setState function (like setEmail or setName), the component will indeed re-render. However, the values stored in the state (like name and email) do not get lost during this process.

Each piece of state in a functional component is independent. When you call setEmail(newEmail), React updates only the email state. The name state remains unchanged unless you explicitly call setName to change it.

Button in from field is by default is submit type, onclicking button it submits form, if you do not write handleSubmit form by default send collected data to currect server.

So write own handleSubmit and put preventDefault() to stop mentioned default behaviour

import React, { useState } from 'react';

function App() {

  // State management for form inputs

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

  const [email, setEmail] = useState('');

  const [error, setError] = useState('');

  const handleNameChange = (e) => {

    setName(e.target.value);

  };

  const handleEmailChange = (e) => {

    setEmail(e.target.value);

  };

  // Form submission

  const handleSubmit = (e) => {

    e.preventDefault(); // Prevents the default form submission behavior

    if (!name || !email) {

      setError('Both fields are required');

      return;

    }

    setError('');

    console.log('Form submitted with:', { name, email });

    setName('');

    setEmail('');

  };

  return (

    <div>

      <h1>Simple Form Example</h1>

      <form onSubmit={handleSubmit}>

        <div>

          <label>

            Name:

            <input type="text" value={name} onChange={handleNameChange} />

          </label>

        </div>

        <div>

          <label>

            Email:

            <input type="email" value={email} onChange={handleEmailChange} />

          </label>

        </div>

        {error && <p style={{ color: 'red' }}>{error}</p>}

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

      </form>

    </div>

  );

}

export default App;

# Use of useRef

1. **Access to DOM Elements**: useRef allows you to directly interact with DOM elements by creating a reference that can be attached to the ref attribute.
2. **Retains Mutable Values**: It can hold mutable values (like previous state or prop values) without causing re-renders when those values change.
3. **Non-DOM Values**: useRef is not limited to DOM references; it can hold any type of value, making it useful for storing values that persist between renders.
4. **Passing as Props**: Refs can be passed as props to child components, enabling direct DOM manipulation or access to mutable values from parent components.

import React, { useRef } from 'react';

function App() {

  const inputRef = useRef(null); // Create a ref

  const focusInput = () => {

    inputRef.current.focus(); // Directly focus the input element

  };

  return (

    <div>

      <input ref={inputRef} type="text" placeholder="Click button to focus" />

      <button onClick={focusInput}>Focus Input</button>

    </div>

  );

}

export default App;

Spread operator  
  
const arr1 = [1, 2, 3];

const arr2 = [arr1, 4, 5]; // Result: arr2 = [[1, 2, 3], 4, 5];

const arr2 = [...arr1, 4, 5]; // Result: arr2 = [1, 2, 3, 4, 5];

setItems((prevItems) => [...prevItems, newItem]);

# Context Api

/context/TodoItemContext.jsx

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

export const TodoItemContext = createContext();

export const useTodo = () => useContext(TodoItemContext);

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

  const [todo, setTodo] = useState([]);

  const addTodo = (item) => {

    setTodo((prev) => [...prev, item]);

  };

  const removeTodo = (itemToDel) => {

    setTodo((prev) => prev.filter((item) => item !== itemToDel));

  };

  return (

    <TodoItemContext.Provider value={{ todo, addTodo, removeTodo }}>

      {children}

    </TodoItemContext.Provider>

  );

};

/Todo.jsx

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

import {TodoItemContext, useTodo} from './context/TodoItemContext';

const Todo = () => {

    const { todo, addTodo, removeTodo } = useTodo();

    const [newItem, setnewItem] = useState("");

    return (

        <div>

            <input type="text" placeholder='Enter Item to add' value={newItem} onChange={(e) => setnewItem(e.target.value)} />

            <button onClick={()=>{addTodo(newItem)}}>add</button>

            <ul>

                {

                    todo.map((item, index) => {

                        return <li key={index}>{item} <button onClick={()=>{removeTodo(item)}}>delete</button></li>

                    })

                }

            </ul>

        </div>

    )

}

export default Todo

App.jsx

import React from 'react';

import { TodoItemProvider } from './context/TodoItemContext';

import Todo from './Todo';

function App() {

  return (

    <TodoItemProvider>

      <Todo />

    </TodoItemProvider>

  );

}

export default App;

# useReducer

import React, { useReducer } from 'react';

import { cartReducer, initialState } from './cartReducer';

function ShoppingCart() {

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

  const addItem = (item) => {

    dispatch({ type: 'add', payload: item });

  };

  const removeItem = (item) => {

    dispatch({ type: 'remove', payload: item });

  };

  return (

    <div>

      <h1>Shopping Cart</h1>

      <p>Total Items: {state.totalItems}</p>

      <button onClick={() => addItem({ id: 1, name: 'Apple' })}>Add Apple</button>

      <button onClick={() => addItem({ id: 2, name: 'Banana' })}>Add Banana</button>

      <h2>Items in Cart:</h2>

      <ul>

        {state.items.map(item => (

          <li key={item.id}>

            {item.name}

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

          </li>

        ))}

      </ul>

    </div>

  );

}

export default ShoppingCart;

// cartReducer.js

export const initialState = {

    items: [],

    totalItems: 0,

  };

  export function cartReducer(state, action) {

    if (action.type === 'add') {

      return {

        ...state,

        items: [...state.items, action.payload],

        totalItems: state.totalItems + 1,

      };

    } else if (action.type === 'remove') {

      const updatedItems = state.items.filter(item => item.id !== action.payload.id);

      return {

        ...state,

        items: updatedItems,

        totalItems: updatedItems.length,

      };

    } else {

      return state;

    }

  }

# Some doubts

 return {

        ...state,

        items: [...state.items, action.payload],

        totalItems: state.totalItems + 1,

      };

So the final returned object will **not** look like this:

{

    items: [], // This would be incorrect as it gets overridden

    totalItems: 0, // This would be incorrect as it also gets overridden

    items: [...state.items, action.payload], // This will override the previous items

    totalItems: state.totalItems + 1, // This will override the previous totalItems

  };

The final returned state would look like this after an item is added:

{

  items: [...], // This includes all previous items plus the new one

  totalItems: 1 // Incremented value based on the previous totalItems

}

You can indeed use the following return statement directly:

return {

    items: [...state.items, action.payload],

    totalItems: state.totalItems + 1,

  };

By using ...state, you ensure that any additional properties in the state object are preserved. This is particularly important if your state object evolves over time. For example, if you later add more properties to the state, such as totalPrice or discount, these properties would be retained without requiring changes in your reducer.

Using items: [...state.items, action.payload] instead of items: [state.items, action.payload] is important

For example, if state.items is ['Apple', 'Banana'] and action.payload is 'Orange', using the spread operator will create:

items: ['Apple', 'Banana', 'Orange']

If you write items: [state.items, action.payload], it creates a new array with two elements:

items: [['Apple', 'Banana'], 'Orange']

# useEffect

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

function App() {

  const [data, setData] = useState([]);

  useEffect(() => {

    const fetchData = async () => {

      const response = await fetch('https://jsonplaceholder.typicode.com/posts');

      const result = await response.json();

      setData(result);

    };

    fetchData();

  }, []);

  return (

    <div>

      <h1>Posts</h1>

        <ul>

          {data.map(post => (

            <li key={post.id}>{post.title}</li>

          ))}

        </ul>

    </div>

  );

}

export default App;

useEffect(() => {

  console.log('This runs only once on mount');

}, []); // Runs once after initial render

useEffect(() => {

  console.log('This runs after every render');

}); // Runs after every render (not recommended for performance reasons)

useEffect(() => {

  console.log('Count has changed:', count);

}, [count]); // Runs after initial render and whenever `count` changes

# The useEffect Hook Cleanup

**Purpose of Cleanup**: The cleanup function in useEffect is crucial for managing side effects in React components, particularly for actions that require cleanup when the component unmounts or before the effect runs again. This is essential for preventing memory leaks and ensuring proper resource management (like clearing timers or removing event listeners).

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

function TimerComponent() {

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

  useEffect(() => {

    const timerID = setInterval(() => {

      setCount(prevCount => prevCount + 1); // Increment the count every second

    }, 1000); // 1000 ms = 1 second

    // Cleanup function to clear the interval

    return () => {

      clearInterval(timerID);

    };

  }, []); // Empty dependency array to run the effect only once on mount

  return <div>Count: {count}</div>;

}

export default TimerComponent;

# useCallback

If ParentComponent re-renders due to the count state changing, the ChildComponent will not re-render because the reference to handleButtonClick remains the same, preventing unnecessary rendering.

The useCallback hook in React is used to memoize functions, meaning it preserves the function across re-renders.

It ensures that child components receive the same function reference

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

function ChildComponent({ onButtonClick }) {

  console.log('Child component rendered');

  return <button onClick={onButtonClick}>Click Me</button>;

}

function ParentComponent() {

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

  // Memoizing the callback function with useCallback

  const handleButtonClick = useCallback(() => {

    console.log('Button clicked!');

  }, []); // This function will not change between renders

  console.log('Parent component rendered');

  return (

    <div>

      <h1>Count: {count}</h1>

      <button onClick={() => setCount(count + 1)}>Increment Count</button>

      <ChildComponent onButtonClick={handleButtonClick} />

    </div>

  );

}

export default ParentComponent;

# useMemo

useCallback memorizes functions, useMemo memorizes values

useMemo caches the result of expensive calculations so that the same computation is not repeated on every render.

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

function ExpensiveCalculationComponent() {

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

  // An expensive calculation that only runs when `count` changes

  const expensiveCalculation = useMemo(() => {

    console.log('Calculating expensive value...');

    let total = 0;

    for (let i = 0; i < 1000000000; i++) {

      total += i; // Simulating an expensive calculation

    }

    console.log('Completed')

    return total;

  }, [count]); // Only recalculates when `count` changes

  return (

    <div>

      <h1>Count: {count}</h1>

      <button onClick={() => setCount(count + 1)}>Increment Count</button>

      <p>Expensive Calculation Result: {expensiveCalculation}</p>

    </div>

  );

}

export default ExpensiveCalculationComponent;

# Outlet

Outlet is a component that serves as a placeholder for rendering child routes. It allows you to define a nested routing structure where certain components can render based on the current route.

import React from 'react';

import {

  BrowserRouter as Router,

  Routes,

  Route,

  Link,

  Outlet

} from 'react-router-dom';

// Main Layout

function Layout() {

  return (

    <div>

      <h1>My Website</h1>

      <nav>

        <Link to="/">Home</Link> | <Link to="/about">About</Link> | <Link to="/contact">Contact</Link>

      </nav>

      <hr />

      <Outlet /> {/\* Renders the matched child route \*/}

    </div>

  );

}

// Home Component

function Home() {

  return <h2>Home Page</h2>;

}

// About Component

function About() {

  return <h2>About Page</h2>;

}

// Contact Component

function Contact() {

  return <h2>Contact Page</h2>;

}

// App Component

function App() {

  return (

    <Router>

      <Routes>

        <Route path="/" element={<Layout />}>

          <Route index element={<Home />} /> {/\* Default child route \*/}

          <Route path="about" element={<About />} />

          <Route path="contact" element={<Contact />} />

        </Route>

      </Routes>

    </Router>

  );

}

export default App;

# React Query Overview

import { useQuery, useMutation, useQueryClient } from "@tanstack/react-query";

function App() {

  // Setting up a query to fetch data

  const { data, error, isLoading } = useQuery({

    queryKey: ["todo"],

    queryFn: () =>

      fetch("https://jsonplaceholder.typicode.com/posts").then((res) =>

        res.json()

      ),

  });

  // Access the query client to manage cache

  const queryClient = useQueryClient();

  // Mutation setup for posting data

  const { mutate, isPending, isError, isSuccess } = useMutation({

    mutationFn: (newPost) =>

      fetch("https://jsonplaceholder.typicode.com/posts", {

        method: "POST",

        body: JSON.stringify(newPost),

        headers: { "Content-type": "application/json; charset=UTF-8" },

      }).then((res) => res.json()),

    onSuccess: () => {

      queryClient.invalidateQueries(["todo"]); // Refreshes data on success

    },

  });

  if (isLoading) return <p>Loading...</p>;

  if (error) return <p>Error: {error.message}</p>;

  return (

    <div>

      <h1>Posts</h1>

      <ul>

        {data?.map((post) => (

          <li key={post.id}>{post.title}</li>

        ))}

      </ul>

      <button onClick={() => mutate({ title: "New Post" })}>Add Post</button>

      {isPending && <p>Adding post...</p>}

      {isSuccess && <p>Post added successfully!</p>}

      {isError && <p>Error adding post.</p>}

    </div>

  );

}

export default App;

It's a library that helps you manage the state of data you fetch from servers, like APIs, in your React applications.  
*One of the most powerful tools for managing server-side state in React.*

**Advantages**

* **Data Fetching Made Easy**: With a simple useQuery hook, fetching data becomes super easy.
* **Built-in Loading and Error States**: No need to write custom code for handling loading, errors, or success states.
* **Automatic Caching**: React Query automatically caches your data.
* **Background Refetching**: If your data gets stale or out of date, TanStack Query can refetch it in the background.
* **Pagination and Infinite Scrolling**: Handling pagination or infinite scrolling? React Query has tools specifically designed for these complex use cases.

**Core Functions**

* **refetch**: Re-fetches data for a specific query, typically triggered manually to get the latest data (e.g., refetch() in useQuery).
* **invalidateQueries**: Marks queries as "stale" so they automatically refetch next time they're accessed. Useful after data changes (e.g., queryClient.invalidateQueries("todo")).
* **resetQueries**: Resets query state (like loading and error states) to its initial state without refetching data (e.g., queryClient.resetQueries("todo")).
* **removeQueries**: Deletes a query from the cache entirely, clearing its data and state (e.g., queryClient.removeQueries("todo")).

**Configuration Options**

* **staleTime**: Controls how long data is served as "fresh" from the cache without re-fetching.
* **cacheTime**: Controls how long the query data stays in the cache after no longer being used.

**Example Scenario**  
If staleTime is set to 5 minutes, data will be served from the cache for up to 5 minutes without re-fetching.  
If cacheTime is set to 10 minutes, data will stay in the cache for 10 minutes after it’s no longer in use, after which it will be removed from memory.