React Hooks

What is hook?

Hooks are the functions to use some react features in functional components.

In other words, Hooks are functions that make Functional components work like class components.

Before hooks there was only one way to use state and lifecycle methods

by using the class components

```
import React, { Component } from "react";

class Counter extends Component {
  constructor(props) {
    super(props);
    this.state = { count: 0 }; // Defining state inside the constructor
  }

  increment = () => {
    this.setState({ count: this.state.count + 1 }); // Updating state using setState
  };

  componentDidMount() {
    console.log("Component Mounted"); // Lifecycle method
  }

  componentDidUpdate() {
    console.log("Component Updated"); // Lifecycle method
  }

  componentWillUnmount() {
    console.log("Component Will Unmount"); // Cleanup before component is rem
  }

  render() {
    return (
      <div>
        <h2>Counter: {this.state.count}</h2>
        <button onClick={this.increment}>Increment</button>
      </div>
    );
  }
}

export default Counter;
```

Hooks are functions that make Functional components work like Class components

What is useState hook?

useState hook is a function used to add state in functional component.

state is nothing but just values or variables of your component.

State is also known as data that changes with time or across renders in React

useState hook returns an array with two elements , 1st element is our original value and 2nd is a function

```
import "./App.css";
import { useState } from "react";

function App() {
  let [counter, setCounter] = useState(0);
  let [inputText, setInputText] = useState("");
  // let array = useState(0);
  // let counterValue = array[0];
  // let setCounterValue = array[1];

  function increaseCounter() {
    setCounter(counter + 1);
  }

  function handleChange(event) {
    console.log(event.target.value);
  }

  return (
    <div className="App">
      <h1>Counter:{counter}</h1>
      <button onClick={increaseCounter}>Increase</button>
      <h1>Input text</h1>
      <input type="text" onChange={handleChange} />
      <h1>inputText value {inputText}</h1>
      <input
        type="text"
        onChange={(e) => {
          setInputText(e.target.value);
        }}
      />
    </div>
  );
}

export default App;
```

Another example

```
import "./App.css";
import { useState } from "react";

function App() {
  let [details, setDetails] = useState({
    counter: 0,
    inputText: "",
  });

  function handleChange(type, event) {
    setDetails((prevDetails) => ({
      ...prevDetails,
      [type]: type === "counter" ? prevDetails.counter + 1 : event.target.value,
    }));
  }

  return (
    <div className="App">
      <h1>Counter: {details.counter}</h1>
      <button onClick={(e) => handleChange("counter", e)}>Increase</button>

      <h1>Input text</h1>
      <input type="text" onChange={(e) => handleChange("inputText", e)} />

      <h2>Typed Text: {details.inputText}</h2>
    </div>
  );
}

export default App;
```

what is useEffect hook ?

useEffect is used to perform side effects in our component

side effects are actions which are performed with the outside world

we perform a side effect when we need to reach outside of our React components to do something example :- fetching data from api

updating the dom document and window

timer functions , set timeout and set interval

useEffect hook accepts two arguments

1.callback

2.dependencies

useEffect is combination of componentDidMount , componentDidUpdate and componentWillUnmount

Variation of useEffect

1. useEffect without dependencies
2. useEffect with empty array
3. useEffect with variables

1.useEffect without dependencies will run every time component renders

here the title change on every single render

if we pass empty array then useEffect runs only one time when our component gets rendered

if we add dependency then it will run whenever dependency value changes

```
import "./App.css";
import { useEffect, useState } from "react";
```

```
function App() {
  const [counter, setCounter] = useState(0);

  useEffect(() => {
    document.title = `Counter: ${counter}`;
  });

  // useEffect(() => {
  //  document.title = `Counter: ${counter}`;
  //},[]);

  // useEffect(() => {
  //  document.title = `Counter: ${counter}`;
  //},[counter]);

  return (
    <div className="App">
      <h1>Counter: {counter}</h1>
      <button onClick={() => setCounter(counter + 1)}>Increment</button>
    </div>
  );
}

export default App;
```

Clean up function in useEffect

```
import "./App.css";
import { useEffect, useState } from "react";

function App() {
  const [counter, setCounter] = useState(0);
  const [time, setTime] = useState(0);
  useEffect(() => {
    console.log("render");
    setInterval(() => {
      setTime((time) => time + 1);
    }, 1000);
  });

  return (
    <div className="App">
      <h1> time :{time}</h1>
    </div>
  );
}

export default App;
```

The issue with your code lies in the useEffect hook. Specifically, the setInterval function is being called repeatedly on every render, which causes multiple intervals to be created. This results in the time state being updated multiple times per second, instead of once every second.

Why this happens:

1. **useEffect without dependencies**:
    - The useEffect hook is missing a dependency array (`[]`). Without it, the effect runs after every render of the component.
    - Each time the component renders, a new setInterval is created, leading to multiple intervals running simultaneously.
2. **No cleanup for setInterval**:
    - The setInterval function is not being cleared when the component re-renders or unmounts. This causes the intervals to pile up, further compounding the issue.
3. **Multiple setInterval instances**:

- If the component is unmounted and remounted (e.g., during hot reloading in development or due to some other logic), the useEffect will run again, creating a new setInterval without clearing the previous one.
- This results in multiple intervals running simultaneously, causing the time state to increment faster than expected.

4. **No cleanup for setInterval**:
    - The setInterval function is not being cleared when the component unmounts or before the effect runs again. This is why the intervals stack up.

Basically it keeps on running so its important to run a clean up function so that our application doesn't run unexpected code

```
import "./App.css";
import { useEffect, useState } from "react";

function App() {
  const [counter, setCounter] = useState(0);
  const [time, setTime] = useState(0);

  useEffect(() => {
    console.log("render");
    const timer = setInterval(() => {
      setTime((time) => time + 1);
    }, 1000);

    return () => {
      clearInterval(timer); // cleanup function
    };
  }, []);

  return (
    <div className="App">
      <h1> time :{time}</h1>
    </div>
  );
}

export default App;
```

clean up function works after you click on the button in the example given below previous useEffect’s clean up happens

```
import "./App.css";
import { useEffect, useState } from "react";

function App() {
  const [counter, setCounter] = useState(0);

  useEffect(() => {
    console.log("useEffect called", counter);
    return () => {
      console.log("clean up", counter);
    };
  }, [counter]);

  return (
    <div className="App">
      <h3>useEffect example</h3>
      <button onClick={() => setCounter(counter + 1)}>Increment</button>
    </div>
  );
}

export default App;

output:useEffect called 19
App.js:10 clean up 19
App.js:8 useEffect called 20
App.js:10 clean up 20
App.js:8 useEffect called 21
```

what is useContext ?

UseContext hook is used to manage global data in react application like global state , services , themes , user settings

we use context hook to access props or state value without passing through multiple child levels

```
import "./App.css";
import { useEffect, useState, createContext } from "react";
import ChildComponent from "./ChildComponent";

const CounterContext = createContext();
function App() {
  return (
    <div className="App">
      <CounterContext.Provider value={{ count: 100, increment: () => {} }}>
        <h1>Parent Component</h1>
        <ChildComponent />
      </CounterContext.Provider>
    </div>
  );
}

export { CounterContext };
export default App;
```

```
import React, { useContext } from "react";
import { CounterContext } from "./App";
```

```
const ChildComponent = () => {
  const { count, increment } = useContext(CounterContext);

  return (
    <div>
      <h1>Child Component</h1>
      <h2>Count: {count}</h2>
      <button onClick={increment}>Increment</button>
    </div>
  );
};

export default ChildComponent;
```

what is useRef?

useRef allows us to access DOM elements

used for creating mutable variables which will not re-render the component

here is an example

lets say we have a state name and input box which has onChange function we want to see how many times render happens

if we use state for count then the value keeps on increasing

```
import "./App.css";
import { useEffect, useRef, useState } from "react";

function App() {
  const [name, setName] = useState("");
  // const [count, setCount] = useState(0);
  const count = useRef(0);

  //if i use it like this then count will keep on increasing
  // useEffect(() => {
  //   setCount(count + 1);
  // });

  useEffect(() => {
    count.current = count.current + 1;
  });

  return (
    <div className="App">
      <input
        type="text"
        value={name}
        onChange={(e) => setName(e.target.value)}
      />
      <h1>name:{name}</h1>
      {/* <h1>render:{count}</h1> */}
      <h1>render:{count.current}</h1>
    </div>
  );
}

export default App;
```

another use case of ref is that we can access direct DOM element

```
import "./App.css";
import { useEffect, useRef, useState } from "react";

function App() {
  const inputRef = useRef();
  const handleClick = () => {
    console.log("function run");
    inputRef.current.style.width = "300px";
  };

  return (
    <div className="App">
      <input ref={inputRef} type="text" />
      <button onClick={handleClick}>Click here</button>
    </div>
  );
}

export default App;
```

What is useReducer?

useReducer is used to manage state in our react application

useReducer works like a state management tool

state management is used to manage all states of application in a simple way

always use the useReducer hook when you have a lot of states and methods to handle

```
import "./App.css";
import { useReducer } from "react";

const initialState = {
  count: 0,
};
const reducer = (state, action) => {
  switch (action.type) {
    case "increment":
      return { ...state, count: state.count + 1 };
    case "decrement":
      return { ...state, count: state.count - 1 };
    default:
      return state;
  }
};

function App() {
  const [state, dispatch] = useReducer(reducer, initialState);

  return (
    <div className="App">
      <h1>count:{state.count}</h1>
      <button onClick={() => dispatch({ type: "increment" })}>Increment</button>
      <button onClick={() => dispatch({ type: "decrement" })}>Decrement</button>
    </div>
  );
}

export default App;
```

Optimization and Performance:

- **Avoiding Unnecessary Re-renders:**When using `useReducer`, you can dispatch actions to update the state, and the component will only re-render when the state actually changes, which can improve performance, especially in large applications.
- **Passing Dispatch Down:**Instead of passing callbacks, you can pass the `dispatch` function down to child components, allowing them to trigger state updates without needing to know the underlying state logic.

Sharing State Across Components:

- **React Context:**

  While `useReducer` is primarily used within a single component, you can combine it with `useContext` to share state and actions across multiple components.

- **Example:**

A shopping cart application where different components need to access and modify the cart's contents. `useReducer` can manage the cart's state, and `useContext` can make the state and actions available to other components.

another example :

Form State Management (Multi-Step Form)

- Keeps track of form steps and user inputs dynamically.
- **Example Components:** `SignupForm`, `ProfileSetup`.

```
const formReducer = (state, action) => {
  switch (action.type) {
    case "UPDATE_FIELD":
      return { ...state, [action.field]: action.value };
    case "NEXT_STEP":
      return { ...state, step: state.step + 1 };
    case "PREV_STEP":
      return { ...state, step: state.step - 1 };
    default:
      return state;
  }
};

const SignupForm = () => {
  const [state, dispatch] = useReducer(formReducer, { step: 1, name: "", email: "

  return (
    <div>
      <p>Step {state.step}</p>
      {state.step === 1 && (
        <input
          type="text"
          placeholder="Name"
          onChange={(e) => dispatch({ type: "UPDATE_FIELD", field: "name", v
        />
      )}
      {state.step === 2 && (
        <input
          type="email"
          placeholder="Email"
          onChange={(e) => dispatch({ type: "UPDATE_FIELD", field: "email", va
        />
      )}
      <button onClick={() => dispatch({ type: "NEXT_STEP" })}>Next</button>
      <button onClick={() => dispatch({ type: "PREV_STEP" })}>Back</button>
    </div>
  );
};
```

what is useLayoutEffect?

useLayoutEffect works exactly the same as useEffect but difference is when it runs

useEffect runs after the DOM is printed on the browser

useLayoutEffect runs before the DOM is printed on the browser

Whenever we want to run code before the DOM is printed

height, width, layout related

useLayoutEffect runs Synchronously

The most common use case of useLayoutEffect is to get the dimension of the layout

work flow

1.React calculate component

2.useLayoutEffect

3.React prints all elements

4.useEffect

so here in the example we want to get dimension of the text and then add padding according to its heights dimension

but if we use useEffect then dom is printed 1st and then text is displayed after that we get dimensions and then padding is added

but if we use useLayoutEffect then we get dimension 1st , add padding and then print the dom

so there is no lag in the UI

```
import "./App.css";
import { useEffect, useLayoutEffect, useRef, useState } from "react";

function App() {
  const [toggle, setToggle] = useState(false);
  const textRef = useRef();

  //if we use useEffect here there is a lag in displaying the text
  //at the correct position
  // because useEffect runs after the DOM is painted
  // useEffect(() => {
  //   if (textRef.current != null) {
  //     const dimensions = textRef.current.getBoundingClientRect();
  //     console.log(dimensions);
  //     textRef.current.style.paddingTop = `${dimensions.height}px`;
  //   }
  // });

  useLayoutEffect(() => {
    if (textRef.current != null) {
      const dimensions = textRef.current.getBoundingClientRect();
      console.log(dimensions);
      textRef.current.style.paddingTop = `${dimensions.height}px`;
    }
  });

  return (
    <div className="App">
      <h1>useLayout example</h1>
      <button onClick={() => setToggle(!toggle)}>Toggle</button>
      {toggle && <p ref={textRef}>This is a paragraph that will be toggled.</p>}
    </div>
  );
}

export default App;
```

`useLayoutEffect` is essential when you need to **synchronize UI updates with layout changes** before the browser repaints the screen. It helps prevent flickering, measure elements, manage animations, and apply styles dynamically.

what is useMemo?

useMemo hook is used to apply Memoization in React.

Memoization is a technique for improving the performance of a code

It is useful to avoid expensive calculations on every render when the returned value is not changed

lets take an example where we have two variables number and dark

we have a expensive function , whenever we change number or toggle the theme , expensive function gets calculated again , which causes lag on the screen

```
import "./App.css";
import { useEffect, useState } from "react";
```

```
function App() {
  const [number, setNumber] = useState(0);
  const [dark, setDark] = useState(false);

  const calculate = expensiveFunction(number);
  const cssStyle = {
    backgroundColor: dark ? "black" : "white",
    color: dark ? "white" : "black",
  };
  return (
    <div className="App">
      <input
        type="number"
        value={number}
        onChange={(e) => setNumber(parseInt(e.target.value))}
      />
      <button onClick={() => setDark((prev) => !prev)}>Change Theme</button>
      <div style={cssStyle}>{calculate}</div>
    </div>
  );
}

function expensiveFunction(num) {
  console.log("Calculating...");
  for (let i = 0; i < 1000000000; i++) {}
  return num * 2;
}

export default App;
```

useMemo syntax is similar to useEffect only difference is we can return value in useMemo so after using useMemo the expensive function will not calculate if we click on toggle theme button

```
import "./App.css";
import { useEffect, useMemo, useState } from "react";

function App() {
  const [number, setNumber] = useState(0);
  const [dark, setDark] = useState(false);

  const memoCalculation = useMemo(() => {
    return expensiveFunction(number);
  }, [number]);

  // const calculate = expensiveFunction(number);
  const cssStyle = {
    backgroundColor: dark ? "black" : "white",
    color: dark ? "white" : "black",
  };
  return (
    <div className="App">
      <input
        type="number"
        value={number}
        onChange={(e) => setNumber(parseInt(e.target.value))}
      />
      <button onClick={() => setDark((prev) => !prev)}>Change Theme</button>
      <div style={cssStyle}>{memoCalculation}</div>
    </div>
  );
}

function expensiveFunction(num) {
  console.log("Calculating...");
  for (let i = 0; i < 1000000000; i++) {}
  return num * 2;
}

export default App;
```

🔷 Key Differences

Feature	`useMemo`	`useEffect`
Purpose	Memoizes **computed values**	Runs **side effects**
Execution	**During render**	**After render**
Returns	Memoized value	Nothing
Use Cases	Expensive calculations, derived state	API calls, subscriptions, DOM updates
Dependencies	Recomputes only when dependencies change	Runs effect when dependencies change

🚀 When to Use What?

- **Use `useMemo`** when you need to optimize expensive calculations inside the render process.
- **Use `useEffect`** when you need to perform side effects like data fetching, subscriptions, or manually changing the DOM.

Example: Optimizing Expensive Computations

Imagine you're building an employee dashboard where you filter and sort a large dataset of employees. If the filtering and sorting logic is expensive, you don't want it to run on every render unless the dependencies change.

```
import React, { useState, useMemo } from "react";

const EmployeeList = ({ employees }) => {
  const [query, setQuery] = useState("");
  // Memoize the filtered and sorted list so it only recomputes when `employees`
  const filteredEmployees = useMemo(() => {
    console.log("Computing filtered employees...");
    return employees
      .filter(emp => emp.name.toLowerCase().includes(query.toLowerCase()))
      .sort((a, b) => a.name.localeCompare(b.name));
  }, [employees, query]);

  return (
    <div>
      <input
        type="text"
        placeholder="Search employees..."
        value={query}
        onChange={(e) => setQuery(e.target.value)}
      />
      <ul>
        {filteredEmployees.map(emp => (
          <li key={emp.id}>{emp.name}</li>
        ))}
      </ul>
    </div>
  );
};
```

Example You’re passing an object as a prop to a child component, and the child is wrapped in React.memo(). Without useMemo, the object reference changes on every render, causing unnecessary re-renders.

Without useMemo (Inefficient)

```
import React, { useState } from "react";
import ChildComponent from "./ChildComponent";
```

```
const ParentComponent = () => {
  const [count, setCount] = useState(0);

  const user = { name: "John Doe" }; // New object created on every render

  return (
    <div>
      <button onClick={() => setCount(count + 1)}>Increment {count}</button>
      <ChildComponent user={user} />
    </div>
  );
};

export default ParentComponent;
```

```
import React from "react";

const ChildComponent = React.memo(({ user }) => {
  console.log("Child re-rendered");
  return <p>User: {user.name}</p>;
});

export default ChildComponent;
```

Issue: Even though user is the same, the child re-renders every time because the user object reference changes on every render.

Optimized

```
import React, { useState, useMemo } from "react";
import ChildComponent from "./ChildComponent";

const ParentComponent = () => {
  const [count, setCount] = useState(0);
  const user = useMemo(() => ({ name: "John Doe" }), []); // Memoized object

  return (
    <div>
      <button onClick={() => setCount(count + 1)}>Increment {count}</button>
      <ChildComponent user={user} />
    </div>
  );
};

export default ParentComponent;
```

so you need to use useMemo   in child component as well as for the object

what is useCallback?

useCallback is used to return Memoize function , Its also useful for preventing functions from being re-created on re-rendering.

In the given example , when we toggle the theme react re renders , thats why PrintTable component also re renders

```
import "./App.css";
import { useState } from "react";
import PrintTable from "./PrintTable";

function App() {
  const [number, setNumber] = useState(0);
  const [dark, setDark] = useState(false);

  // const calculate = expensiveFunction(number);
  const cssStyle = {
    backgroundColor: dark ? "black" : "white",
    color: dark ? "white" : "black",
  };

  const calculateTable = () => {
    return [
      number * 1,
      number * 2,
      number * 3,
      number * 4,
      number * 5,
      number * 6,
      number * 7,
      number * 8,
      number * 9,
      number * 10,
    ];
  };
  return (
    <div className="App" style={cssStyle}>
      <input
        type="number"
        value={number}
        onChange={(e) => setNumber(parseInt(e.target.value))}
      />
      <PrintTable calculateTable={calculateTable} />
      <button onClick={() => setDark((prev) => !prev)}>Change Theme</button>
    </div>
  );
}

export default App;

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

const PrintTable = ({ calculateTable }) => {
  const [rows, setRows] = useState([]);
  useEffect(() => {
    console.log("Print Table Runs!");
    setRows(calculateTable());
  }, [calculateTable]);

  //map through the table rows and display them
  return rows.map((row, index) => {
    return <div key={index}>{row}</div>;
  });
};

export default PrintTable;
```

useMemo returns memoize value and useCallback returns memoized function

```
import "./App.css";
import { useCallback, useState } from "react";
import PrintTable from "./PrintTable";

function App() {
  const [number, setNumber] = useState(0);
  const [dark, setDark] = useState(false);

  // const calculate = expensiveFunction(number);
  const cssStyle = {
    backgroundColor: dark ? "black" : "white",
    color: dark ? "white" : "black",
  };

  const calculateTable = useCallback(() => {
    return [
      number * 1,
      number * 2,
      number * 3,
      number * 4,
      number * 5,
      number * 6,
      number * 7,
      number * 8,
      number * 9,
      number * 10,
    ];
  }, [number]);

  return (
    <div className="App" style={cssStyle}>
      <input
        type="number"
        value={number}
        onChange={(e) => setNumber(parseInt(e.target.value))}
      />
      <PrintTable calculateTable={calculateTable} />
      <button onClick={() => setDark((prev) => !prev)}>Change Theme</button>
    </div>
  );
}

export default App;


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

const PrintTable = ({ calculateTable }) => {
  const [rows, setRows] = useState([]);

  useEffect(() => {
    console.log("Print Table Runs!");
    setRows(calculateTable());
  }, [calculateTable]);
  //map through the table rows and display them
  return rows.map((row, index) => {
    return <div key={index}>{row}</div>;
  });
};

export default PrintTable;
```

now print table console.log will not be displayed

To **completely prevent `PrintTable` from re-rendering when only the theme changes**, wrap it in `React.memo()`

Optimizing Event Handlers in Large Lists

Scenario:

You have a large list, and each item has an event handler. Creating a **new function for each item** on every render is inefficient.

Example: Handling Click Events in a Large List

```
import React, { useState, useCallback } from "react";

const LargeList = ({ items }) => {
  // Using useCallback to prevent recreating handleClick on every render
  const handleClick = useCallback((id) => {
    console.log("Clicked item:", id);
  }, []);

  return (
    <ul>
      {items.map((item) => (
        <li key={item.id} onClick={() => handleClick(item.id)}>
          {item.name}
        </li>
      ))}
    </ul>
  );
};

export default LargeList;
```

Why `useCallback()` ?

Without `useCallback`, a new function is created for **each item** on every render.

With `useCallback`, the `handleClick` reference remains the same, improving performance in large lists.

Optimizing Function Props in `useEffect()`

Scenario:

A function is used inside `useEffect()`, and since functions are re-created on every render, the effect runs unnecessarily.

Example: Fetching Data Only When Needed

```
import React, { useState, useEffect, useCallback } from "react";

const FetchData = () => {
  const [data, setData] = useState(null);

  // Memoizing fetchData so useEffect doesn’t re-run unnecessarily
  const fetchData = useCallback(async () => {
    const response = await fetch("https://jsonplaceholder.typicode.com/todos/1");
    const result = await response.json();
    setData(result);
  }, []); // ✅ Function reference remains the same

  useEffect(() => {
    fetchData();
  }, [fetchData]); // ✅ fetchData reference is stable

  return <pre>{JSON.stringify(data, null, 2)}</pre>;
};

export default FetchData;
```

Why `useCallback()` ?

Without `useCallback()` , `fetchData` gets recreated every render, causing `useEffect` to run again unnecessarily.

With `useCallback()` , `fetchData` has a stable reference, so `useEffect` runs **only once** (unless dependencies change).

Optimizing Event Handlers in Large Lists

Scenario:

You have a large list, and each item has an event handler. Creating a **new function for each item** on every render is inefficient.

Example: Handling Click Events in a Large List

```
import React, { useState, useCallback } from "react";

const LargeList = ({ items }) => {
  // Using useCallback to prevent recreating handleClick on every render
  const handleClick = useCallback((id) => {
    console.log("Clicked item:", id);
  }, []);

  return (
    <ul>
      {items.map((item) => (
        <li key={item.id} onClick={() => handleClick(item.id)}>
```

```
          {item.name}
        </li>
      ))}
    </ul>
  );
};

export default LargeList;
```

Why `useCallback()`?

Without `useCallback`, a new function is created for **each item** on every render.

With `useCallback`, the `handleClick` reference remains the same, improving performance in large lists.

Using `useCallback()` with `useMemo()` for Optimized Computation

Scenario:

You are using `useMemo()` for caching computed values, but a function inside the computation keeps changing, causing re-renders.

Example: Caching a Filtered List

```
import React, { useState, useMemo, useCallback } from "react";

const FilterList = ({ items }) => {
  const [query, setQuery] = useState("");

  // Memoizing the filter function
  const filterItems = useCallback((item) => item.includes(query), [query]);

  const filteredItems = useMemo(() => items.filter(filterItems), [items, filterItems]);
  return (
    <div>
      <input value={query} onChange={(e) => setQuery(e.target.value)} placeholde
      <ul>
        {filteredItems.map((item, index) => (
          <li key={index}>{item}</li>
        ))}
      </ul>
    </div>
  );
};

export default FilterList;
```

Why `useCallback()` ?

Without `useCallback()` , `filterItems` would be re-created on every render, invalidating `useMemo()` .

With `useCallback()` , `filterItems` has a stable reference, allowing `useMemo()` to properly cache the filtered list.

What is Custom Hook?

Custom hooks are basically a reusable function

custom hooks are your own hooks that you create for your own use and you can use them multiple times in your project.

suppose you have to write a logic and need to use it everywhere you can just write a hook for it and use it

here in the example below

app.js uses a custom hook to fetch data from any url that is passed to the hook

```
import "./App.css";
import useFetch from "./customHooks/useFetch";

function App() {
  const { data, loading } = useFetch(
    "https://jsonplaceholder.typicode.com/posts"
  );

  return (
    <div className="App">
      <h1>Custom hooks</h1>
      {loading ? (
        <h1>Loading...</h1>
      ) : (
        <div>
          {data.map((post) => (
            <div key={post.id}>
              <h3>{post.title}</h3>
              <p>{post.body}</p>
            </div>
          ))}
        </div>
      )}
    </div>
  );
}

export default App;
```

```
import { useState, useEffect } from "react";

const useFetch = (url) => {
  const [data, setData] = useState(null);
  const [loading, setLoading] = useState(true);

  useEffect(() => {
    fetch(url)
      .then((res) => res.json())
      .then((data) => {
        setData(data);
        setLoading(false);
      });
  }, [url]);

  return { data, loading };
};

export default useFetch;
```

How to implement infinite scroll bar ?

To implement an infinite scroll feature in your code, you can use the `IntersectionObserver` API or a library like `react-infinite-scroll-component`. Below is an implementation using the `IntersectionObserver` API:

```
import "./App.css";
import useFetch from "./customHooks/useFetch";
import { useState, useRef, useCallback } from "react";

function App() {
  const [page, setPage] = useState(1); // State to track the current page
  const { data, loading } = useFetch(
    `https://jsonplaceholder.typicode.com/posts?_page=${page}&_limit=10`
  );
```

```
const [posts, setPosts] = useState([]); // State to store all posts
const observer = useRef();

// Callback to handle intersection
const lastPostRef = useCallback(
  (node) => {
    if (loading) return;
    if (observer.current) observer.current.disconnect();
    observer.current = new IntersectionObserver((entries) => {
      if (entries[0].isIntersecting) {
        setPage((prevPage) => prevPage + 1); // Increment the page number
      }
    });
    if (node) observer.current.observe(node);
  },
  [loading]
);

// Update posts when data changes
if (data.length > 0 && posts.length !== page * 10) {
  setPosts((prevPosts) => [...prevPosts, ...data]);
}

return (
  <div className="App">
    <h1>Custom hooks with Infinite Scroll</h1>
    <div>
      {posts.map((post, index) => {
        if (index === posts.length - 1) {
          // Attach the ref to the last post
          return (
            <div ref={lastPostRef} key={post.id}>
              <h3>{post.title}</h3>
              <p>{post.body}</p>
            </div>
          );
          } else {
            return (
              <div key={post.id}>
                <h3>{post.title}</h3>
                <p>{post.body}</p>
              </div>
            );
          }
        })}
      </div>
      {loading && <h1>Loading...</h1>}
    </div>
  );
}

export default App;
```

1. **Pagination**:
    - The API URL is updated to include `_page` and `_limit` query parameters to fetch data page by page.
    - The `page` state tracks the current page number.
2. **IntersectionObserver**:
    - The `lastPostRef` is attached to the last post in the list.
    - When the last post is visible in the viewport, the `page` state is incremented to fetch the next set of posts.
3. **Posts State**:
    - The `posts` state stores all the posts fetched so far.
    - New posts are appended to the existing list whenever new data is fetched.
4. **Loading Indicator**:
    - A loading message is displayed while fetching new posts.