React Hooks Guide with Examples

1. useState

useState is the hook that allows you to add state to a functional component.

It returns a state variable and a function to update that state.

Example 1:

```jsx
import { useState } from 'react';
function Counter() {
    const [count, setCount] = useState(0);
    return (
        <div>
            <p>Count: {count}</p>
            <button onClick={() => setCount(count + 1)}>Increase</button>
        </div>
    );
}
```

Example 2:

```jsx
import { useState } from 'react';
function TextInput() {
    const [text, setText] = useState("");
    return (
```

```
        <div>
            <input type="text" value={text} onChange={(e) => setText(e.target.value)} />
            <p>Text: {text}</p>
        </div>
    );
}
```

useState allows the component to have its own state that can change over time.

2. useEffect

useEffect is the hook that runs side effects in your component.

It is used for actions like fetching data, subscribing to external data sources, or manually changing the DOM.

Example 1:

```jsx
import { useState, useEffect } from 'react';
function DataFetcher() {
    const [data, setData] = useState(null);

    useEffect(() => {
        fetch('https://api.example.com/data')
            .then(response => response.json())
            .then(data => setData(data));
    }, []);  // Empty array means it runs only once (on component mount)
```

```
    return <div>{data ? JSON.stringify(data) : "Loading..."}</div>;
}
```

Example 2:

```jsx
import { useState, useEffect } from 'react';
function Timer() {
    const [time, setTime] = useState(0);

    useEffect(() => {
        const timer = setInterval(() => setTime(time => time + 1), 1000);
        return () => clearInterval(timer);  // Cleanup on unmount
    }, []);  // Runs only once on mount

    return <div>Time: {time}s</div>;
}
```

useEffect is commonly used for data fetching, timers, or changing the DOM manually.

3. useContext

useContext allows you to access values from a React context.

It's typically used for global state or sharing information across components.

Example 1:

```jsx
import { useContext } from 'react';

const ThemeContext = React.createContext('light');


function ThemeSwitcher() {
    const theme = useContext(ThemeContext);
    return <div>Current theme: {theme}</div>;
}
```

Example 2:

```jsx
import { useContext } from 'react';

const UserContext = React.createContext();


function UserInfo() {
    const user = useContext(UserContext);
    return <div>User: {user.name}</div>;
}
```

useContext is great for sharing data like themes, user info, or settings.

4. useRef

useRef is used for accessing a DOM element or keeping a mutable value.

Unlike state, it does not trigger a re-render when changed.

Example 1:

```jsx
import { useRef } from 'react';
function FocusInput() {
    const inputRef = useRef(null);

    const focusInput = () => {
        inputRef.current.focus();
    };

    return (
        <div>
            <input ref={inputRef} type="text" />
            <button onClick={focusInput}>Focus Input</button>
        </div>
    );
}
```

Example 2:

```jsx
import { useRef } from 'react';
function Timer() {
    const intervalRef = useRef(null);
```

```
    const startTimer = () => {
        intervalRef.current = setInterval(() => console.log('Tick'), 1000);
    };

    const stopTimer = () => {
        clearInterval(intervalRef.current);
    };

    return (
        <div>
            <button onClick={startTimer}>Start Timer</button>
            <button onClick={stopTimer}>Stop Timer</button>
        </div>
    );
}
```

useRef is great for accessing DOM elements or handling timers.

5. useReducer

useReducer is used for managing complex state logic in your component.

It's often preferred when there are multiple state values that change together.

Example 1:

```jsx
import { useReducer } from 'react';
```

```
const initialState = { count: 0 };

function reducer(state, action) {
    switch (action.type) {
        case 'increment':
            return { count: state.count + 1 };
        case 'decrement':
            return { count: state.count - 1 };
        default:
            return state;
    }
}

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

    return (
        <div>
            <p>Count: {state.count}</p>
            <button onClick={() => dispatch({ type: 'increment' })}>Increase</button>
            <button onClick={() => dispatch({ type: 'decrement' })}>Decrease</button>
        </div>
    );
}
```

Example 2:

```jsx
import { useReducer } from 'react';

const initialState = { todos: [] };

function reducer(state, action) {
    switch (action.type) {
        case 'add':
            return { todos: [...state.todos, action.payload] };
        default:
            return state;
    }
}

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

    const addTodo = (todo) => {
        dispatch({ type: 'add', payload: todo });
    };

    return (
        <div>
            <button onClick={() => addTodo('New Todo')}>Add Todo</button>
            <ul>
                {state.todos.map((todo, index) => (
```

```
                <li key={index}>{todo}</li>
            ))}
        </ul>
    </div>
  );
}
```

useReducer is useful for managing complex state, especially in large applications.

6. useCallback

useCallback is used to memoize a function, ensuring it is not re-created unless necessary.

This helps to optimize performance, especially when passing functions to child components.

Example 1:

```jsx
import { useState, useCallback } from 'react';
function Parent() {
  const [count, setCount] = useState(0);

  const increment = useCallback(() => setCount(count + 1), [count]);

  return <Child onClick={increment} />;
}

function Child({ onClick }) {
```

```
    return <button onClick={onClick}>Increment</button>;
}
```

Example 2:

```jsx
import { useState, useCallback } from 'react';
function ExpensiveComponent({ compute }) {
    return <div>{compute()}</div>;
}

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

    const compute = useCallback(() => count * 2, [count]);

    return <ExpensiveComponent compute={compute} />;
}
```

useCallback helps avoid unnecessary re-renders by memoizing functions.

7. useMemo

useMemo is used to memoize the result of a function so it is recomputed only when necessary.

This can improve performance by avoiding expensive calculations on each render.

Example 1:

```jsx
import { useMemo, useState } from 'react';
function ExpensiveComponent() {
    const [count, setCount] = useState(0);

    const expensiveValue = useMemo(() => {
        return count * 1000;
    }, [count]);

    return (
        <div>
            <p>Expensive Value: {expensiveValue}</p>
            <button onClick={() => setCount(count + 1)}>Increase</button>
        </div>
    );
}
```

Example 2:

```jsx
import { useMemo, useState } from 'react';
function FilterList() {
    const [filter, setFilter] = useState("");
    const items = ["apple", "banana", "orange", "mango"];

    const filteredItems = useMemo(() => {
```

```
    return items.filter(item => item.includes(filter));
  }, [filter]);

  return (
    <div>
      <input
        type="text"
        value={filter}
        onChange={(e) => setFilter(e.target.value)}
      />
      <ul>
        {filteredItems.map(item => <li key={item}>{item}</li>)}
      </ul>
    </div>
  );
}
```

useMemo is helpful for optimizing expensive calculations or filtering lists.