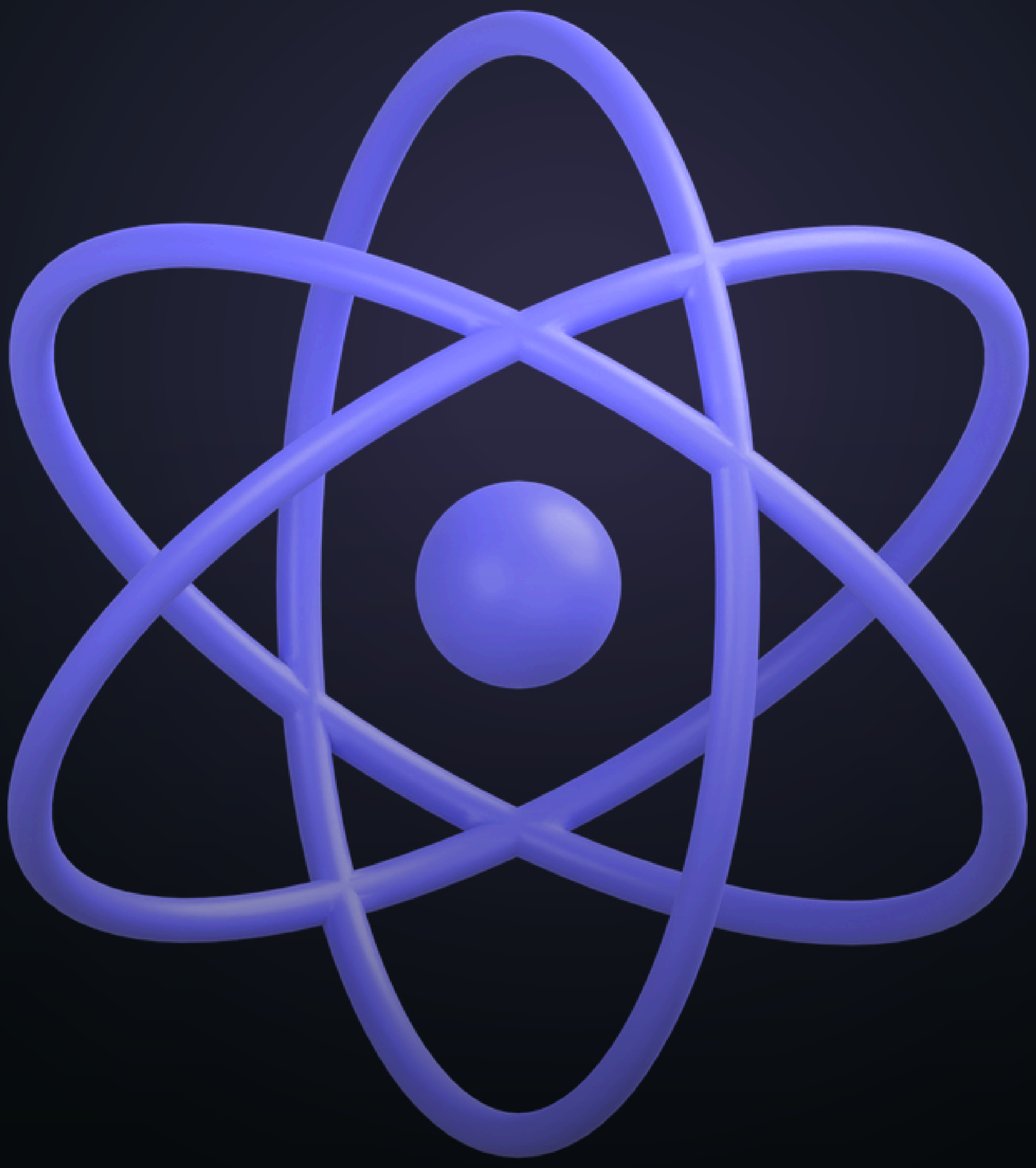
Ram Maheshwari
@rammcodes

React Hooks Cheatsheet

Boost Your React Skills - Hook By Hook!



Swipe

React Hooks allow you to bring state and lifecycle features into functional components, letting you move beyond class components.

With **Hooks**, you can write cleaner, modular code that's easier to test and reuse across your application.

Swipe to level-up and master **React Hooks** 👉

useState - Manage State

The **useState** hook lets you add state to functional components.

In this example, **useState** initializes count at 0. **setCount** updates it when the button is clicked, making state changes easy in functional components.

app.jsx

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

// Initialize count to 0, and setCount is the function to update it
const [count, setCount] = useState(0);

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

Swipe

useEffect - Side Effects

The **useEffect** hook manages side effects like data fetching or DOM manipulation.

In this example, **useEffect** updates the document title whenever count changes. Adding [count] as a dependency ensures it only runs when count changes.

app.jsx

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

useEffect(() => {
  // Sets the document title every time count changes
  document.title = `Count: ${count}`;
}, [count]); // Only re-run the effect if count changes
```

Swipe

useContext - Access Context

The **useContext** hook provides access to context values directly.

In this example, **user** is fetched from **UserContext**, giving easy access to global data like user details without needing extra components.

app.jsx

```
const user = useContext(UserContext);

<p>{user.name}</p>
```

Swipe

useReducer - Complex State Logic

The **useReducer** hook is ideal for managing complex state logic.

This example uses a **reducer** to handle state changes based on actions, making it easier to manage multiple, related state updates.

app.jsx

```
const [count, dispatch] = useReducer(
  (state, action) => action === 'increment' ? state + 1 : state - 1,
  0
);


<button onClick={() => dispatch('increment')}>
  +
</button>
```

Swipe

useRef - Persistent Value and DOM Access

The **useRef** hook lets you persist values or directly access DOM elements.

In this example, the **useRef** hook is used to focus an input element programmatically without triggering a re-render.

app.jsx

```jsx
const inputRef = useRef();

<button onClick={() => inputRef.current.focus()}>Focus</button>
```

Swipe

useMemo - Memoize Expensive Computations

The **useMemo** hook caches computed values to avoid re-calculating on each render.

In this example, **useMemo** memoizes sortedItems, so it only recalculates if items changes, optimizing performance for complex calculations.

app.jsx

```
const sortedItems = useMemo(() => sortItems(items), [items]);
```

Swipe

useCallback - Memoize Functions

The **useCallback** hook caches functions to prevent unnecessary re-renders.

In this example, **useCallback** memoizes handleClick, preventing it from being recreated on every render and saving performance.

app.jsx

```
let memoizedFunction = useCallback(() => handleClick(id), [id]);
```

Swipe

useLayoutEffect - DOM-Based Side Effects

The **useLayoutEffect** hook runs after the DOM updates, but before the browser repaints.

The **useLayoutEffect** hook is useful for reading layout data and synchronizing animations immediately after the DOM changes.

app.jsx

```
useLayoutEffect(() => {
  console.log('Runs after DOM updates');
});
```

Swipe

Custom Hooks - Reusable Logic

The **Custom** hooks encapsulate reusable logic for easy sharing across components.

This **custom** hook called **useFetch** fetches data from a URL, making it reusable in any component with just one line.

app.jsx

```
function useFetch(url) {
  const [data, setData] = useState(null);
  useEffect(() => { fetch(url).then(res => res.json()).then(setData); }, [url]);
  return data;
}

const data = useFetch('/api/data');
```

Swipe

React Hooks simplify functional components by bringing in state and lifecycle features.

Hooks like **useState**, **useEffect**, and **custom** hooks make code modular, clean, and efficient for modern web development 🔥

Swipe

Ram Maheshwari
@rammcodes

Make sure to **Follow** me for more Amazing Content related to Programming & Web Development

Ram Maheshwari

@rammcodes

Go Back