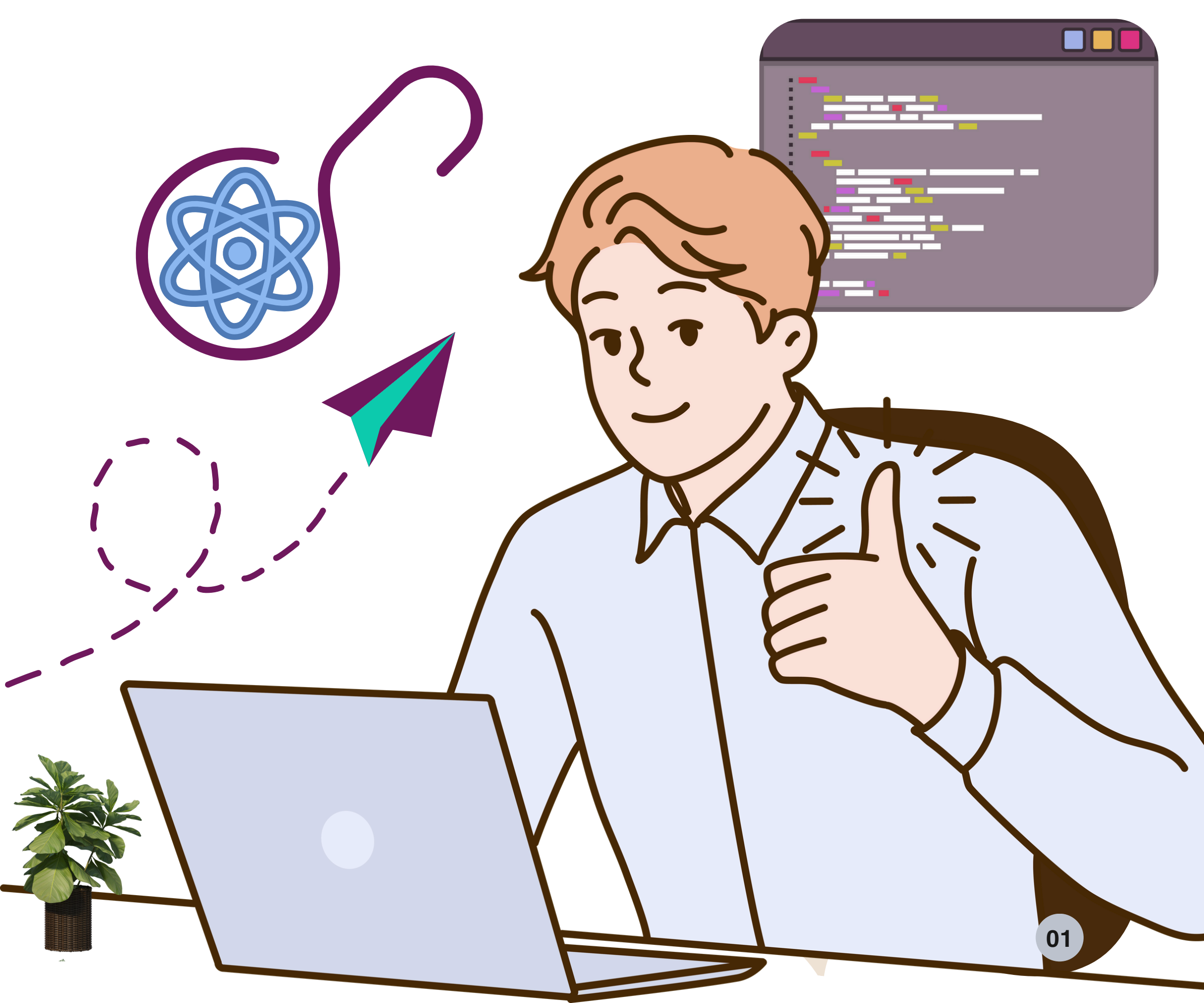
CleanUI

11 CORE REACT HOOKS

TO TAKE YOUR SKILLS TO THE NEXT LEVEL!



CleanUI

1. useState

This React hook allows components to manage local state without class components.

Store simple values like numbers, strings, objects, or arrays.

Example:

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

return (
  <button onClick={() => setCount(count + 1)}>Count: {count}</button>
);
```

Pro Tip: Avoid directly mutating the state. Instead, always update it using '*setState*'.

CleanUI

2. useEffect

useEffect is used for tasks like data fetching, setting up subscriptions, and performing initial actions. It behaves similarly to componentDidMount in class components

Runs after rendering. You can control when it executes by passing dependencies. It functions similarly to shouldComponentUpdate or componentDidUpdate

Example:

```
useEffect(() => {
  document.title = `Count: ${count}`;
}, [count]);  // Runs when `count` changes
```

Pro Tip: Use cleanup functions to avoid memory leaks.

```
useEffect(() => {
  const interval = setInterval(() => console.log("Running"), 1000);
  return () => clearInterval(interval);  // Cleanup
}, []);
```

CleanUI

3. useContext

useContext allows you to access the value of a context directly in a functional component, without needing to use a higher-order component like Consumer

Example:

```
const ThemeContext = createContext("light");

function ThemedButton() {
  const theme = useContext(ThemeContext);
  return <button className={theme}>Click me</button>;
}
```

Pro Tip: If you need advanced state management, combine it with *Redux* or *Zustand*.

CleanUI

4. useRef

It is used to create mutable references to DOM elements and any values that you want to persist across renders without causing re-renders.

Example: Autofocus an input field

```
const inputRef = useRef(null);

useEffect(() => inputRef.current.focus(), []);

return <input ref={inputRef} placeholder="Auto Focused Input" />;
```

```
const count = useRef(0);

const incrementCount = () => {
  count.current += 1;
  console.log(count.current); // Logs updated value, but no re-render happens
};
```

Pro Tip: Useful for handling animations, measuring elements, and persisting values without re-renders.

CleanUI

5. useMemo

Returns a cached value and prevents expensive calculations from running on every render.

Example:

```
const expensiveCalculation = useMemo(() => {
  return computeHeavyTask(value);
}, [value]); // Recalculates only when `value` changes
```

Pro Tip: Use it only when necessary—overusing can degrade performance.

CleanUI

6. useCallback

Memoizes a function, ensuring that the function reference remains stable across re-renders unless its dependencies change. This is especially useful when passing functions down to child components, as it prevents unnecessary re-renders.

Example:

```
const handleClick = useCallback(() => {
  console.log("Clicked!");
}, []);

<button onClick={handleClick}>Click Me</button>;
```

Pro Tip: Use it when passing functions to child components to prevent unnecessary renders.

CleanUI

7. useReducer

- Useful when state management is more complex than a simple state change (like useState)
- Like useState, useReducer is local to the component, but it gives you more flexibility and control over how the state is updated.

Example:

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

const [state, dispatch] = useReducer(reducer, { count: 0 });

return <button onClick={() => dispatch({ type: "increment" })}>Count: {state.count}</button>;
```

Pro Tip: Works great for managing form states and complex UI state logic.

CleanUI

8. useLayoutEffect

- Like *useEffect*, but fires synchronously after the DOM has been updated but before the browser has painted the changes
- useful for animations and layout measurements.

Example:

```
const ResizableComponent = () => {
  const [dimensions, setDimensions] = useState({ width: 0, height: 0 });
  const divRef = useRef(null);

  useLayoutEffect(() => {
    const rect = divRef.current.getBoundingClientRect();
    setDimensions({ width: rect.width, height: rect.height });
  }, []); // Empty dependency array ensures it runs only once on mount
// Add dependencies in array based on your usecase

  return (
    <div ref={divRef} style={{ resize: 'both', overflow: 'auto' }}>
      <p>Width: {dimensions.width}</p>
      <p>Height: {dimensions.height}</p>
    </div>
  );
```

Pro Tip: Avoid blocking the UI—use it only for UI calculations and measurements.

CleanUI

New & Cool React Hooks!

9. useId

useId is a Hook that generates unique, stable IDs that are consistent across re-renders

Example:

```
const Form = () => {
  const inputId = useId();

  return (
    <div>
      <input id={inputId} type="text" />
    </div>
  );
};
```

Pro Tip: Useful in scenarios involving form elements, dynamic content, or any situation requiring guaranteed unique identifiers across re-renders

CleanUI

10. useDeferredValue

Defers updates to prevent UI lag during heavy computations.

It’s similar to a debounce function but without a timeout. It defers the update until React is done processing other tasks.

Example:

```
const [query, setQuery] = useState("");
const deferredQuery = useDeferredValue(query);

useEffect(() => {
  expensiveSearchFunction(deferredQuery);
}, [deferredQuery]);
```

Pro Tip: Use it for search bars, filtering, or UI-heavy components.

CleanUI

11. useTransition

Allows non-blocking UI updates for better user experience. It’s pertcularly useful when you need to update multiple states without rendering immidietly

Example:

```
import React, { useState, useTransition } from 'react';

const [isPending, startTransition] = useTransition();

  const fetchData = () => {
    // Wrap the entire async operation in startTransition
    // It will make isPending true, you don't need another useState for loading
    startTransition(async () => {
      try {
        const response = await fetch('https://jsonplaceholder.typicode.com/posts');
        if (!response.ok) {
          throw new Error('Failed to fetch data');
        }
        const result = await response.json();
        // (optional) Update the state inside another startTransition
        startTransition(() => {
          setData(result);
        });
      } catch (error) {
        console.error('Error fetching data:', error);
      }
    });
};
```

CleanUI

Final Remarks!

React hooks are powerful and should be used wisely for optimal performance.

Which one is your favorite?

Drop your thoughts in the comments!

CONNECT

cleanui.dev

rahulgwebdev.com