Type Of React.Js Hooks

Before React Hooks (React < 16.8), developers were required to write class components in order to take advantage of certain React Features. But now, React Hooks provides a more ergonomic way to build components because we can use stateful logic without changing our component hierarchy.

Types Of Hooks

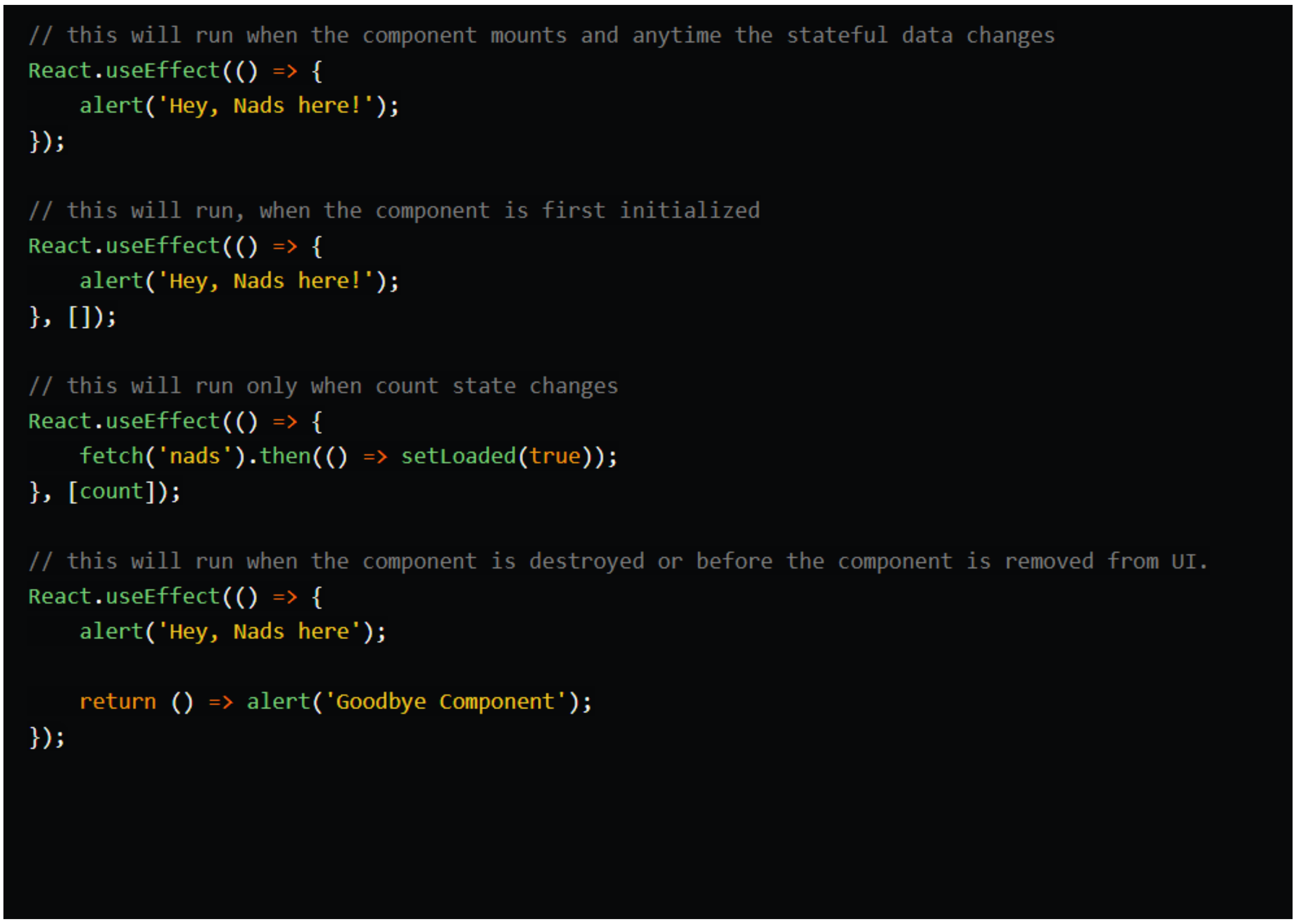
- ### useState

  It is the most important and often used hook. The purpose of this hook is to handle reactive data, any data that changes in the application is called state, when any of the data changes, React re-renders the UI.

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

- ## useEffect :

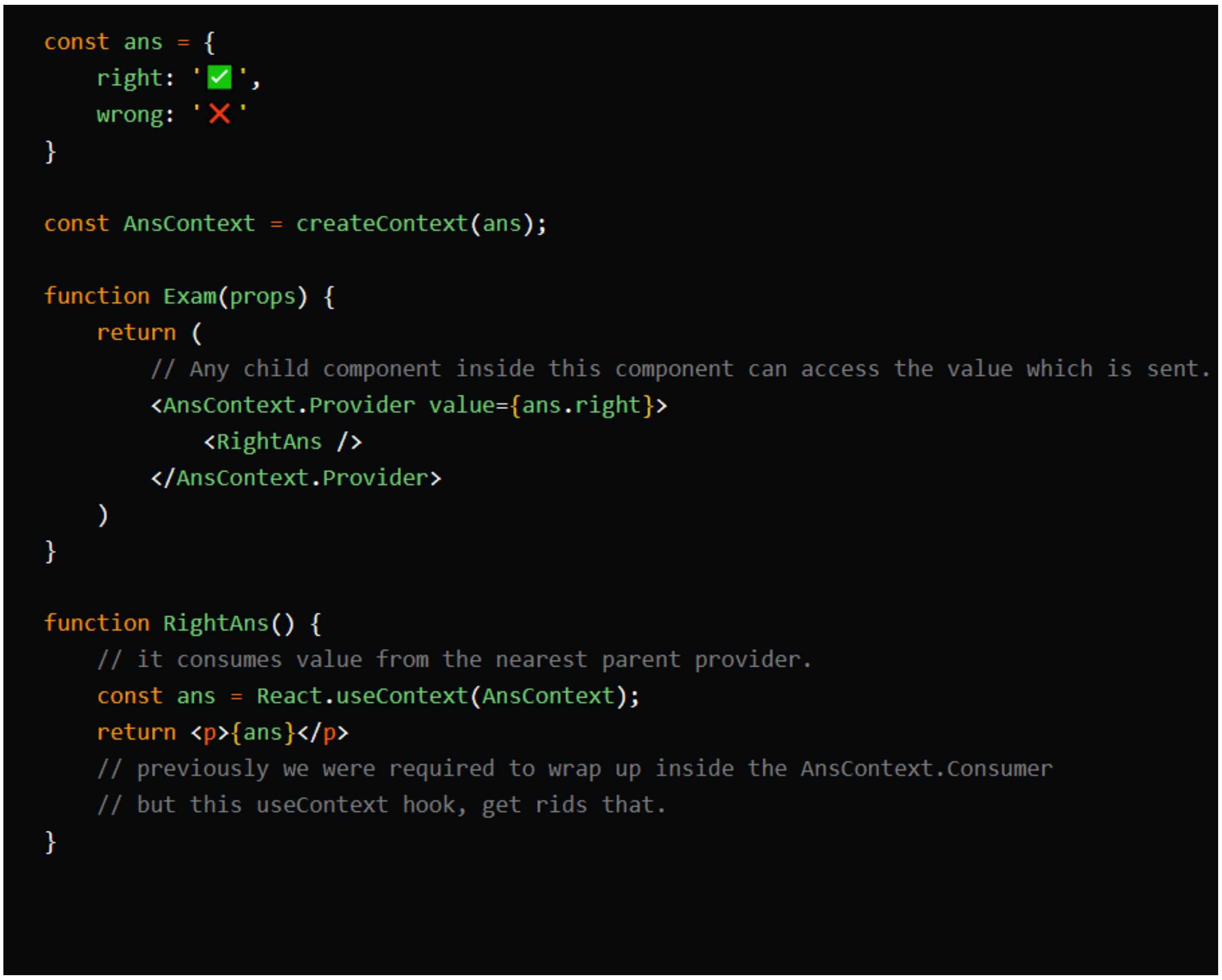
  It allows us to implement all of the lifecycle hooks from within a single function API.



```
// this will run when the component mounts and anytime the stateful data changes
React.useEffect(() => {
    alert('Hey, Nads here!');
});

// this will run, when the component is first initialized
React.useEffect(() => {
    alert('Hey, Nads here!');
}, []);

// this will run only when count state changes
React.useEffect(() => {
    fetch('nads').then(() => setLoaded(true));
}, [count]);

// this will run when the component is destroyed or before the component is removed from UI.
React.useEffect(() => {
    alert('Hey, Nads here');

    return () => alert('Goodbye Component');
});
```

- useContext :

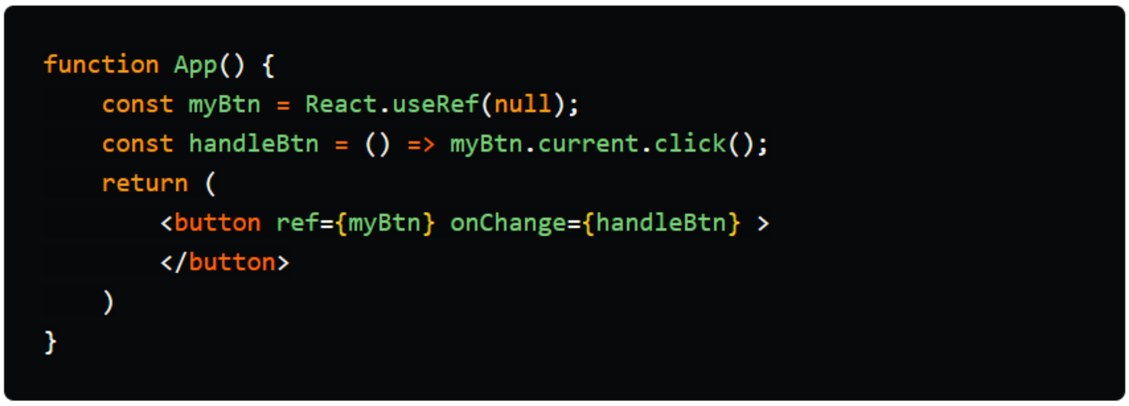
This hook allows us to work with React's Context API, which itself is a mechanism to allow us to share data within its component tree without passing through props. It basically removes **prop-drilling**



```
const ans = {
    right: '✅',
    wrong: '❌'
}

const AnsContext = createContext(ans);

function Exam(props) {
    return (
        // Any child component inside this component can access the value which is sent.
        <AnsContext.Provider value={ans.right}>
            <RightAns />
        </AnsContext.Provider>
    )
}

function RightAns() {
    // it consumes value from the nearest parent provider.
    const ans = React.useContext(AnsContext);
    return <p>{ans}</p>
    // previously we were required to wrap up inside the AnsContext.Consumer
    // but this useContext hook, get rids that.
}
```

• useRef :

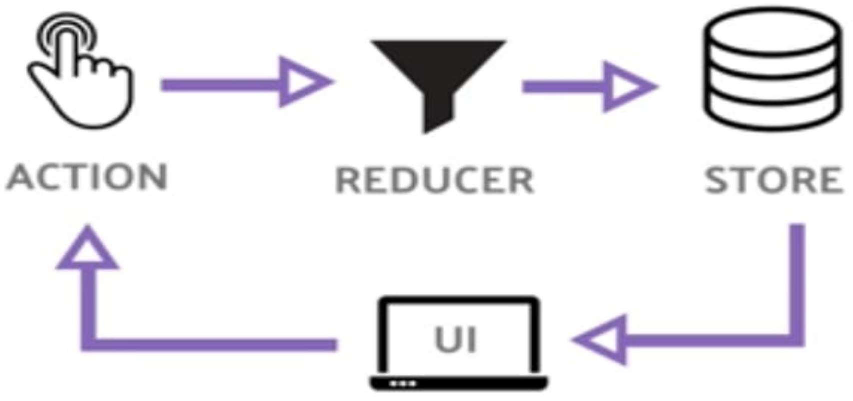
This hook allows us to create a mutable object. It is used, when the value keeps changes like in the case of useState hook, but the difference is, it doesn't trigger a re-render when the value changes. The common use case of this is to grab HTML elements from the DOM.

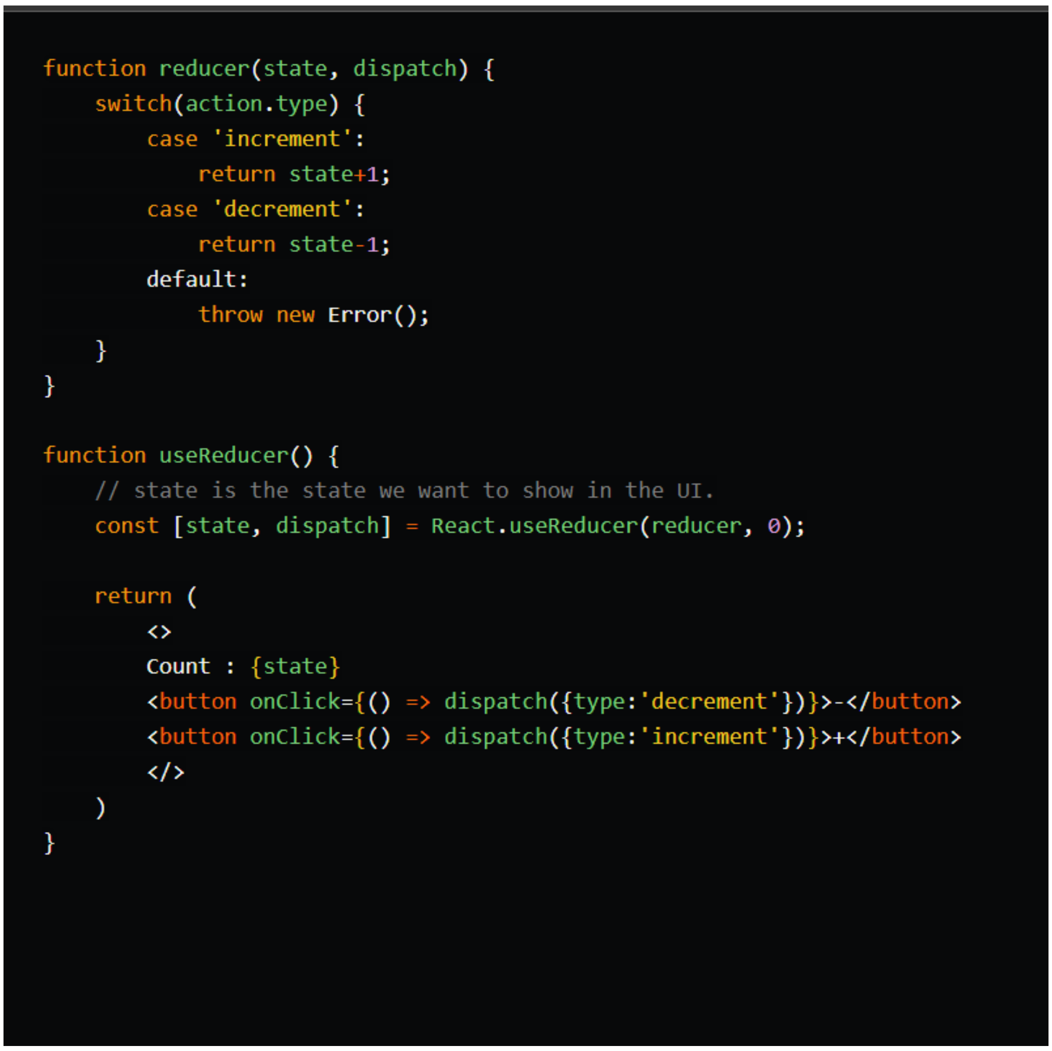


```
function App() {
    const myBtn = React.useRef(null);
    const handleBtn = () => myBtn.current.click();
    return (
        <button ref={myBtn} onChange={handleBtn} >
        </button>
    )
}
```

• useReducer :

It does very similar to setState, It's a different way to manage the state using Redux Pattern. Instead of updating the state directly, we dispatch actions, that go to a reducer function, and this function figures out, how to compute the next state.





```
function reducer(state, dispatch) {
    switch(action.type) {
        case 'increment':
            return state+1;
        case 'decrement':
            return state-1;
        default:
            throw new Error();
    }
}

function useReducer() {
    // state is the state we want to show in the UI.
    const [state, dispatch] = React.useReducer(reducer, 0);

    return (
        <>
        Count : {state}
        <button onClick={() => dispatch({type:'decrement'})}>-</button>
        <button onClick={() => dispatch({type:'increment'})}>+</button>
        </>
    )
}
```