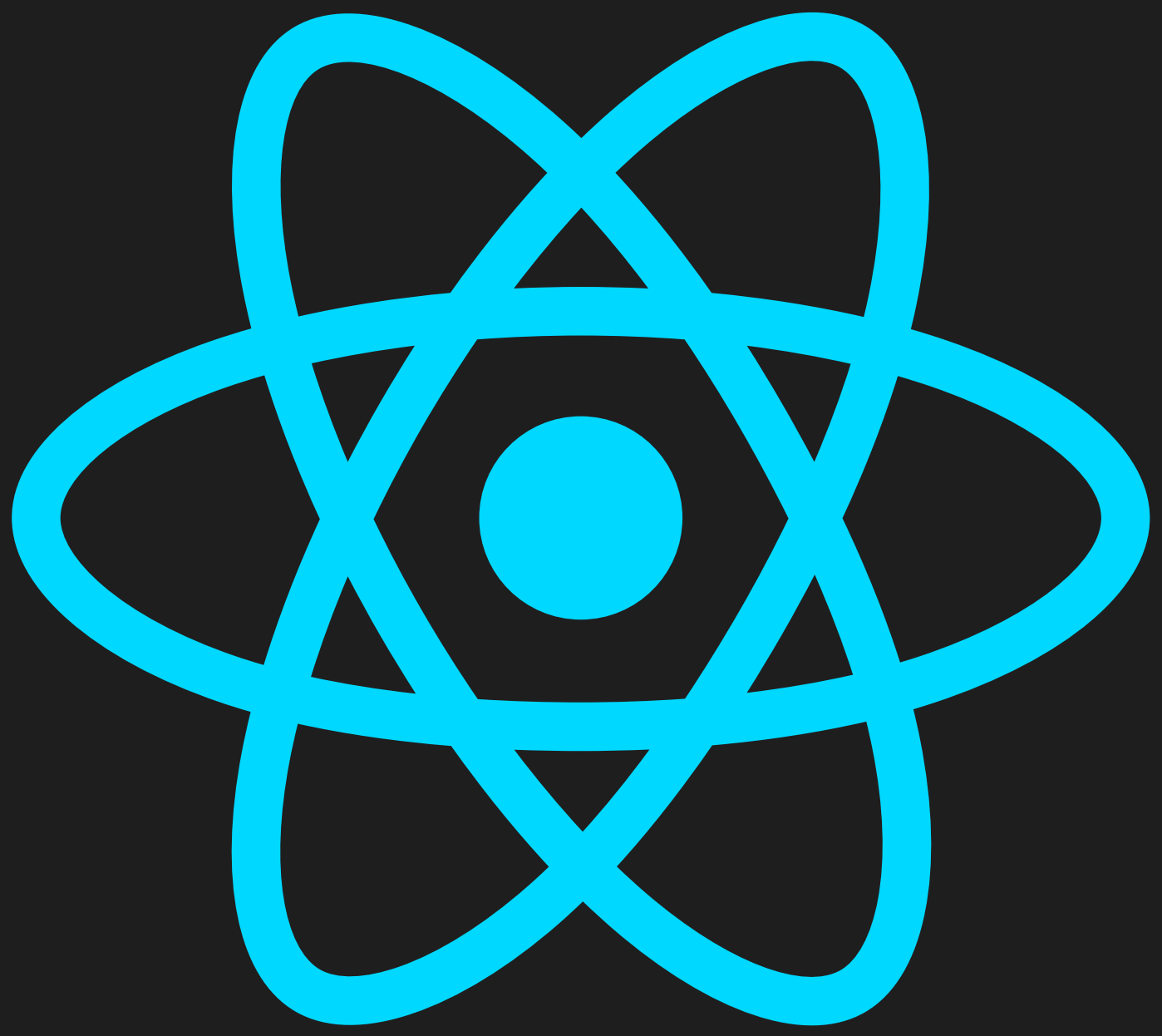
7 React Js HOOKS



1. useState

useState adds state to functional components.

It returns the current state and a function to update it.

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

function Counter() {
  const [count, setCount] = useState(0);
  // more code
```

Here, count starts at 0, and setCount updates it.

2. useEffect

useEffect handles side effects like data fetching or updating the DOM.

It runs after every render, but you can control when it runs by passing dependencies.

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

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

  useEffect(() => {
    document.title = `You clicked ${count} times`;
  }, [count]);
  // more code
```

This updates the document title whenever count changes.

3. useContext

useContext accesses global data or shared state directly, without the need for prop drilling.

```
import React, { useContext } from 'react';

const MyContext = React.createContext();

function DisplayValue() {
  const value = useContext(MyContext);
  // more code
```

This retrieves the value provided by MyContext and displays it.

4. useReducer

useReducer is used for managing complex state logic. It returns the current state and a dispatch function.

```
import React, { useReducer } from 'react';

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

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

This setup helps handle state transitions in a structured way.

5. useRef

useRef allows you to access DOM elements or persist values across renders without causing a re-render.

```
import React, { useRef } from 'react';

function FocusInput() {
  const inputRef = useRef(null);

  const focusInput = () => {
    inputRef.current.focus();
  };
  // more code
```

You can use this reference to directly interact with an input element.

6. useMemo

useMemo optimizes performance by memoizing a calculation, ensuring it only recalculates when dependencies change.

```
import React, { useMemo } from 'react';

function ExpensiveCalculation({ a, b }) {
  const memoizedValue = useMemo(() => {
    return computeExpensiveValue(a, b);
  }, [a, b]);

// more code
```

This caches the result and only recalculates when a or b changes.

7. useCallback

useCallback memoizes functions, preventing them from being recreated unnecessarily.

```
import React, { useCallback } from 'react';

function CallBackApp({ a,b }) {
  const memoizedCallback = useCallback(() => {
    doSomething(a, b);
  }, [a, b]);

  // more code
```

This ensures doSomething only changes when a or b changes.

useCallback differs from useMemo in that useMemo memoizes the result of a computation, while useCallback memoizes the function itself.

FOLLOW