MATIN

14 HOOKS

-ALL REACT- HOOKS

matinscript.com

TOPICS OVERVIEW

useState

When to use

- When you need to add state to a functional component.
- For simple state management.

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

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

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

useEffect

When to use

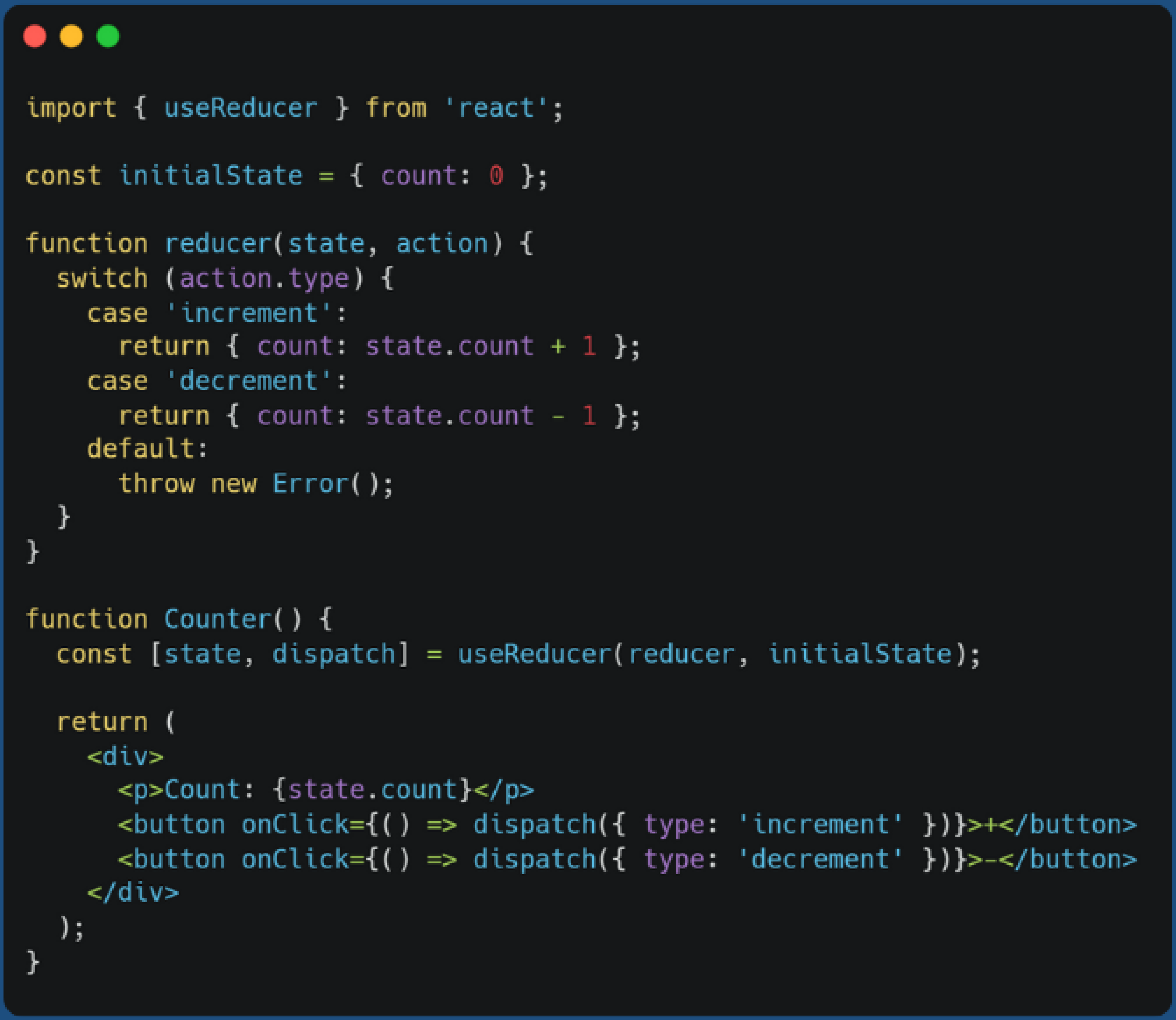
- When you need to perform side effects like data fetching, subscriptions, or manually changing the DOM.
- For running code after render.

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

function Timer() {
  const [seconds, setSeconds] = useState(0);

  useEffect(() => {
    const interval = setInterval(() => {
      setSeconds(prev => prev + 1);
    }, 1000);

    return () => clearInterval(interval);
  }, []);

  return <div>Seconds: {seconds}</div>;
}
```

useReducer

When to use

- When you have complex state logic that involves multiple sub-values or when the next state depends on the previous one.
- For managing state transitions.

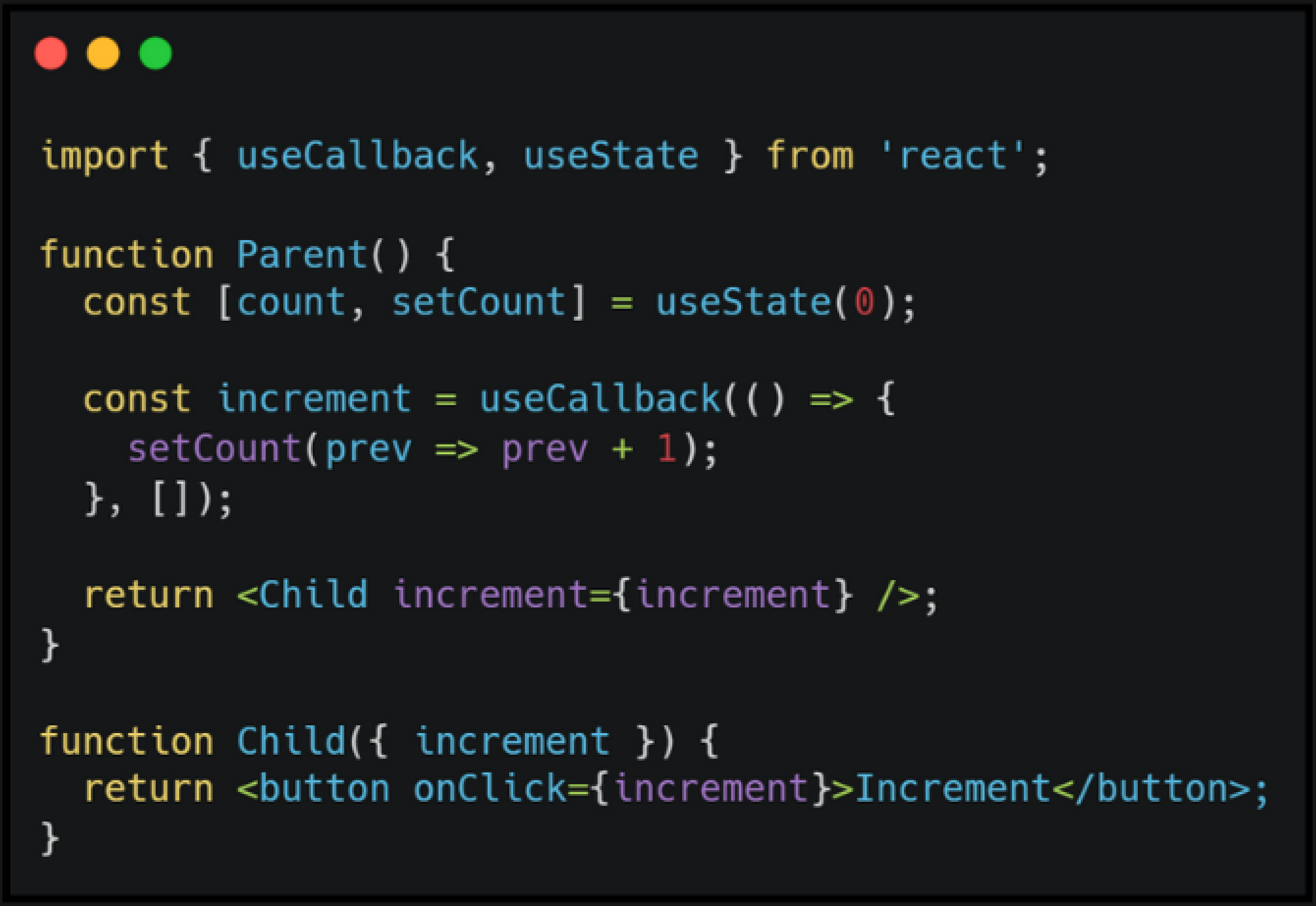


```
import { useReducer } from 'react';

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:
      throw new Error();
  }
}

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

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

useCallback

When to use

- When you need to optimize performance by memoizing functions.
- For preventing unnecessary re-renders.



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

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

  const increment = useCallback(() => {
    setCount(prev => prev + 1);
  }, []);

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

function Child({ increment }) {
  return <button onClick={increment}>Increment</button>;
}
```

useMemo

When to use

- When you need to optimize performance by memoizing expensive calculations.
- For avoiding recalculations on every render.

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

function ExpensiveCalculation({ num }) {
  const result = useMemo(() => {
    return num * 2; // Simulate an expensive calculation
  }, [num]);

  return <div>Result: {result}</div>;
}
```

useContext

When to use

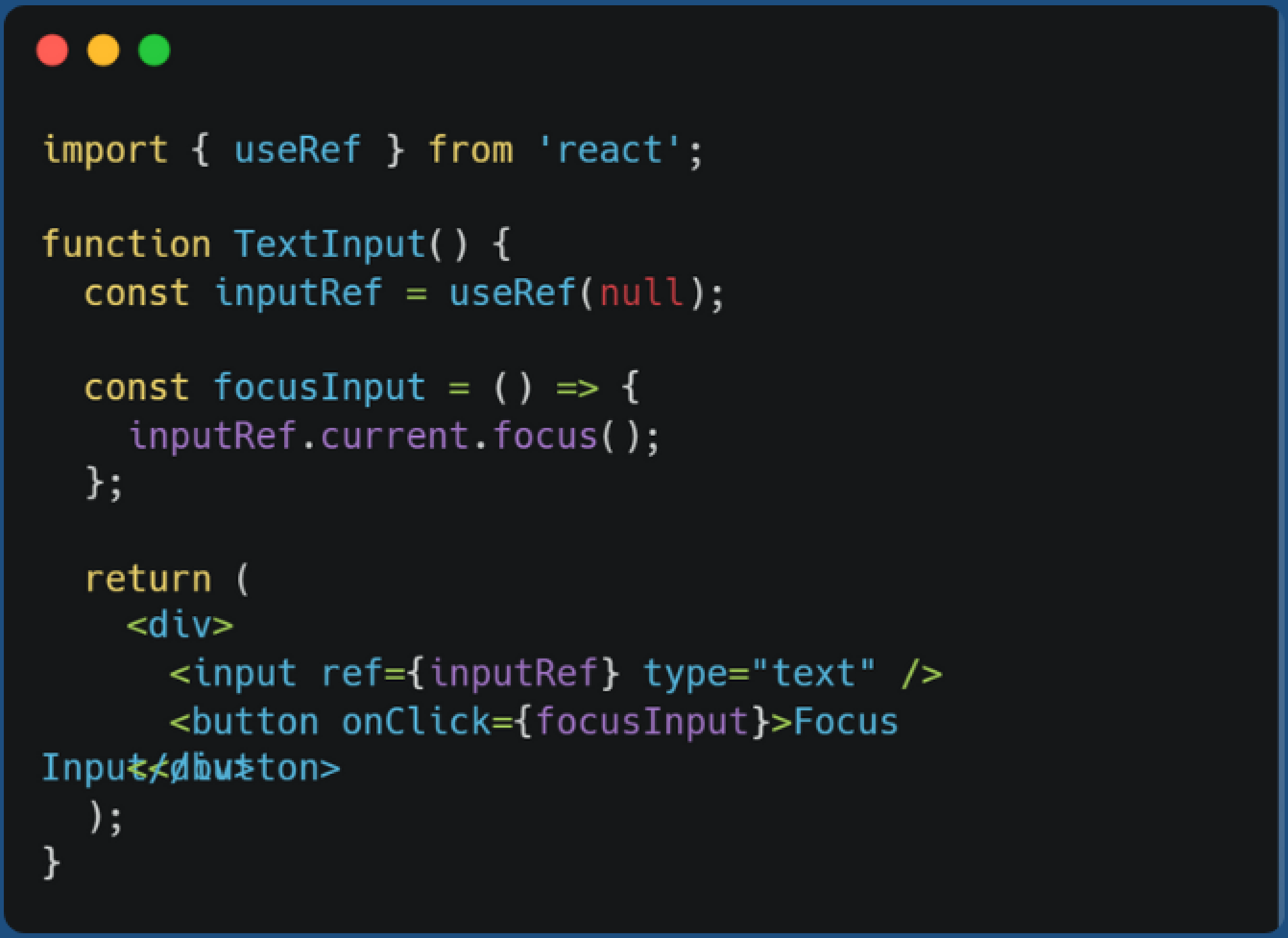
- When you need to consume context values in a functional component.
- For avoiding prop drilling.

```
import { useContext } from 'react';
import { ThemeContext } from './ThemeContext';

function ThemedButton() {
  const theme = useContext(ThemeContext);

  return <button style={{ background: theme.background, color: theme.color
}}> Click me </button>;
}
```

useRef

When to use

- When you need to directly interact with a DOM element.
- For storing mutable values that do not cause re-renders.

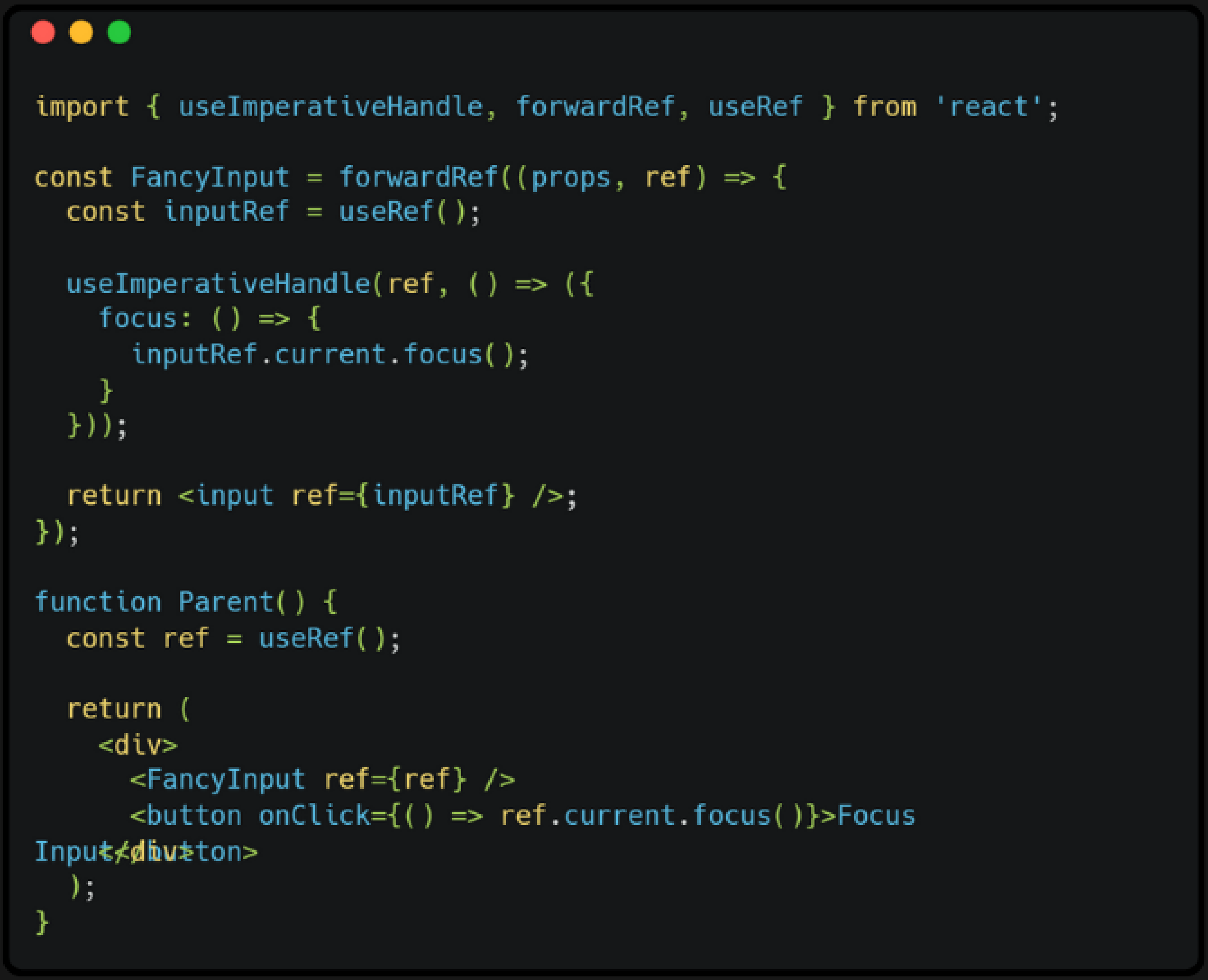


```
import { useRef } from 'react';

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

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

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

useImperativeHandle

When to use

- When you need to customize the ref instance value.
- For exposing imperative methods to parent components.

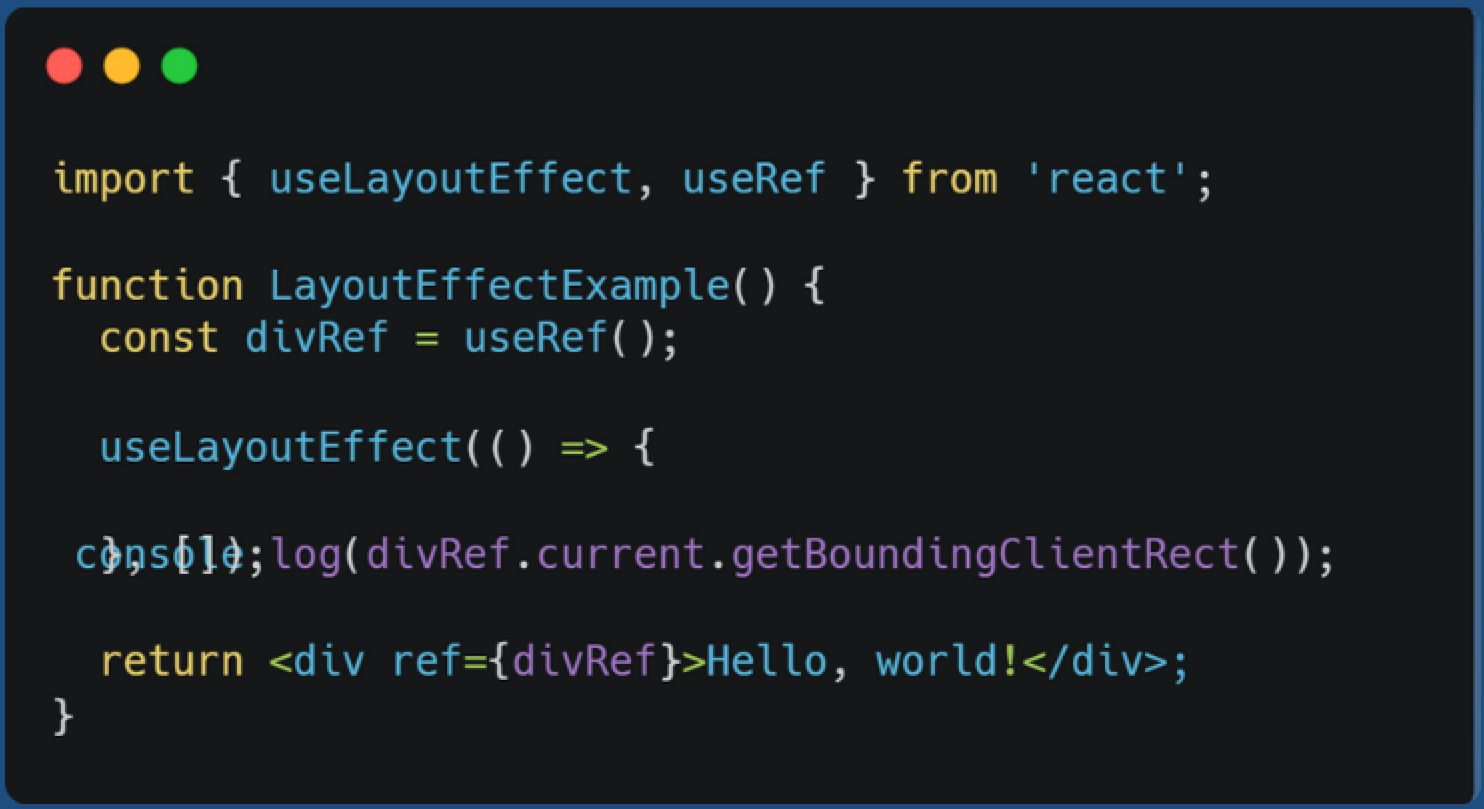


```
import { useImperativeHandle, forwardRef, useRef } from 'react';

const FancyInput = forwardRef((props, ref) => {
  const inputRef = useRef();

  useImperativeHandle(ref, () => ({
    focus: () => {
      inputRef.current.focus();
    }
  }));

  return <input ref={inputRef} />;
});

function Parent() {
  const ref = useRef();

  return (
    <div>
      <FancyInput ref={ref} />
      <button onClick={() => ref.current.focus()}>Focus
Input</button>
    </div>
  );
}
```

useLayoutEffect

When to use

- When you need to read layout from the DOM and synchronously re-render.
- For measuring DOM nodes.

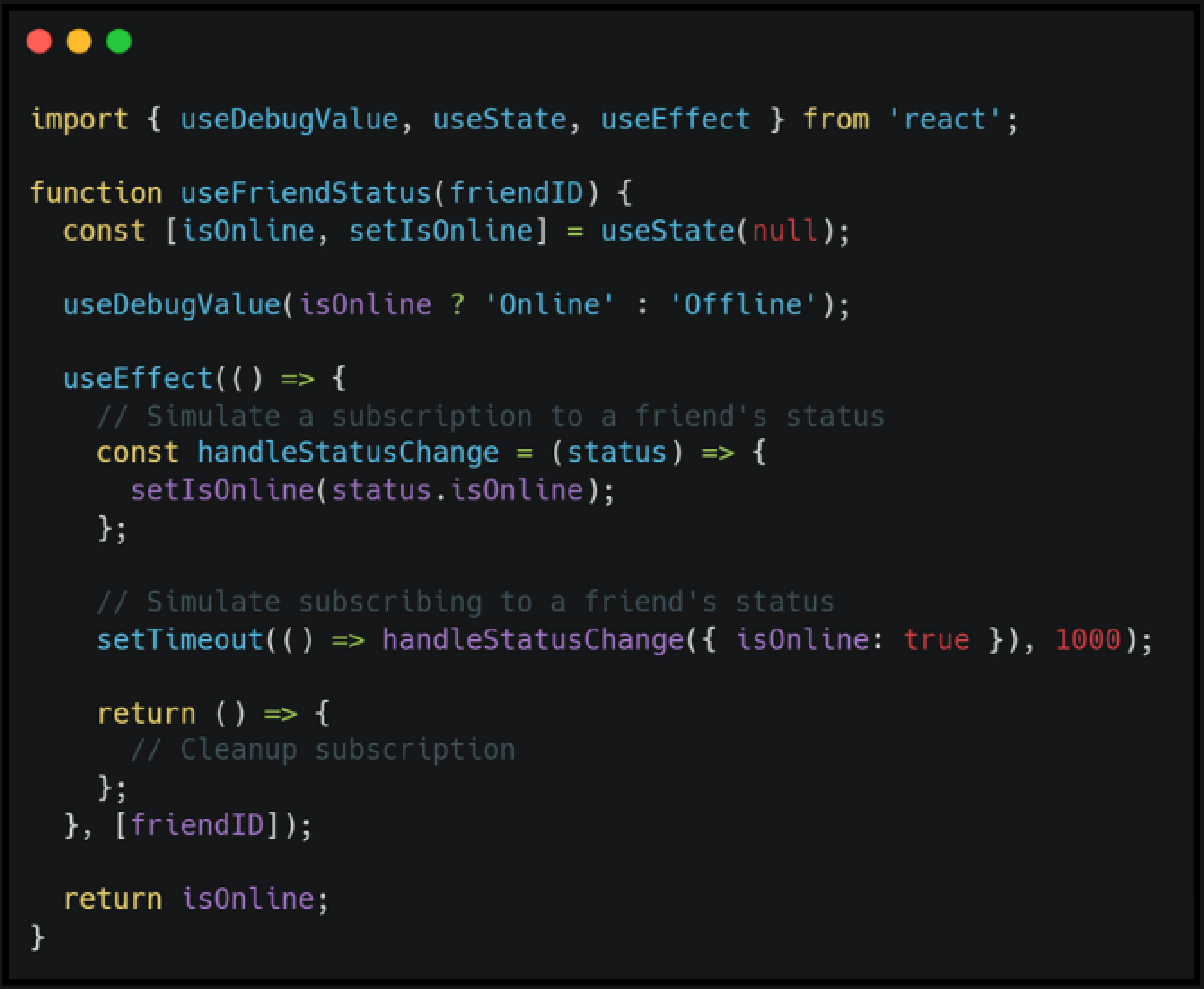


```
import { useLayoutEffect, useRef } from 'react';

function LayoutEffectExample() {
  const divRef = useRef();

  useLayoutEffect(() => {
    console.log(divRef.current.getBoundingClientRect());
  }, []);

  return <div ref={divRef}>Hello, world!</div>;
}
```

useDebugValue

When to use

- When you need to add a label for custom hooks in React DevTools.
- For debugging custom hooks



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

function useFriendStatus(friendID) {
  const [isOnline, setIsOnline] = useState(null);

  useDebugValue(isOnline ? 'Online' : 'Offline');

  useEffect(() => {
    // Simulate a subscription to a friend's status
    const handleStatusChange = (status) => {
      setIsOnline(status.isOnline);
    };

    // Simulate subscribing to a friend's status
    setTimeout(() => handleStatusChange({ isOnline: true }), 1000);

    return () => {
      // Cleanup subscription
    };
  }, [friendID]);

  return isOnline;
}
```

useDeferredValue

When to use

- When you need to defer a value to improve performance.
- For avoiding blocking the main thread.

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

function DeferredValueExample() {
  const [value, setValue] = useState('');
  const deferredValue = useDeferredValue(value);

  return (
    <div>
      <input value={value} onChange={(e) => setValue(e.target.value)}
/>    <p>Deferred Value: {deferredValue}</p>
    </div>
  );
}
```

useTransition

When to use

- When you need to manage state transitions without blocking the UI.
- For handling transitions smoothly.

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

function TransitionExample() {
  const [isPending, start
```

13

useId

When to use

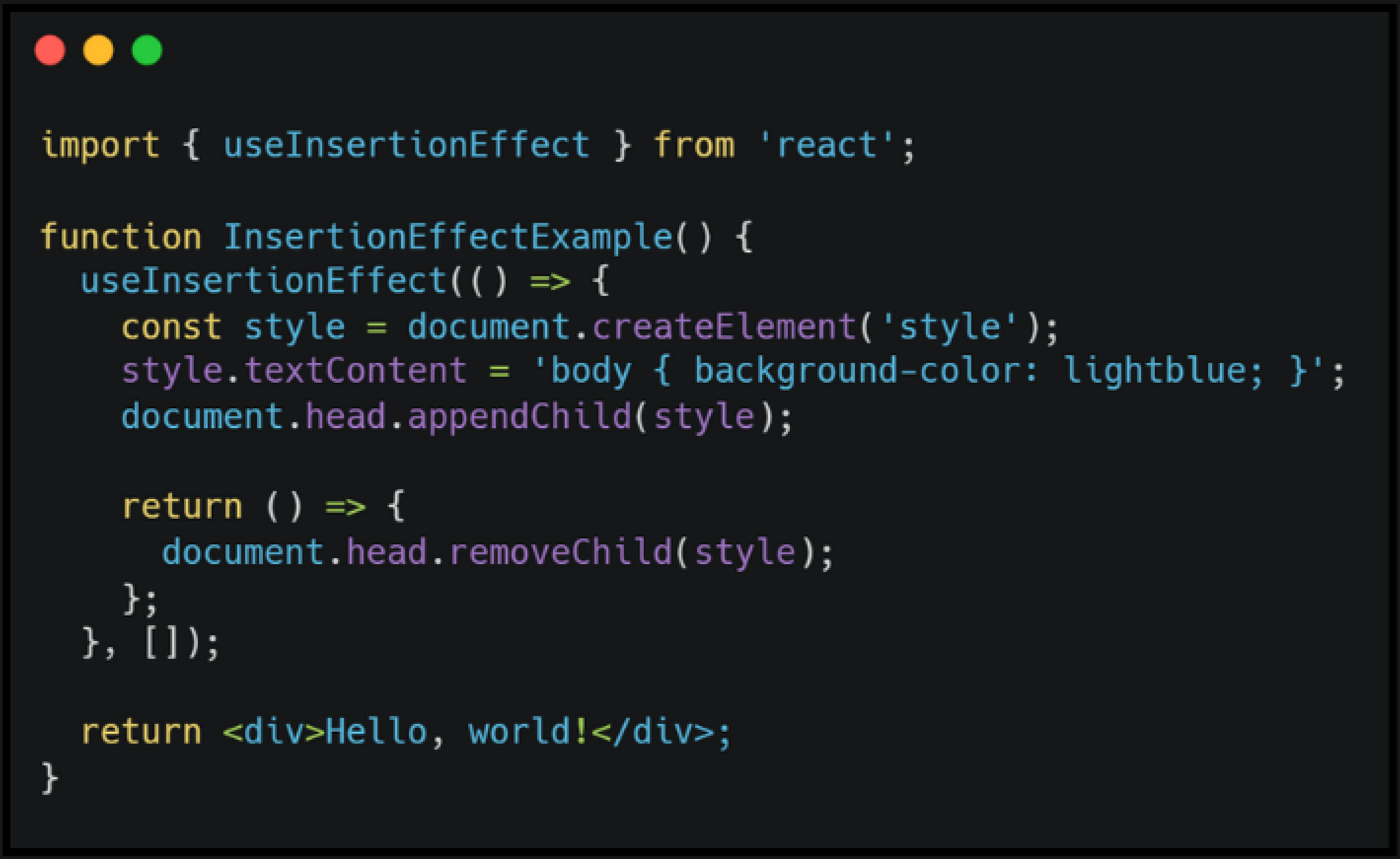
- When you need to generate unique IDs for elements.
- For ensuring accessibility.

```
import { useId } from 'react';

function IdExample() {
  const id = useId();

  return (
    <div>
      <label htmlFor={id}>Name:</label>
      <input id={id} type="text" />
    </div>
  );
}
```

useInsertionEffect

When to use

- When you need to insert styles before DOM mutations.
- For managing styles dynamically.



```
import { useInsertionEffect } from 'react';

function InsertionEffectExample() {
  useInsertionEffect(() => {
    const style = document.createElement('style');
    style.textContent = 'body { background-color: lightblue; }';
    document.head.appendChild(style);

    return () => {
      document.head.removeChild(style);
    };
  }, []);

  return <div>Hello, world!</div>;
}
```

GREAT JOB

Congratulations on mastering React Hooks! Here are some tips and tricks to help you get the most out of them:

Keep It Simple

Start with basic hooks like useState and useEffect before diving into more complex ones.

Custom Hooks

Create custom hooks to encapsulate and reuse logic across your components.

Performance Optimization

Use useMemo and useCallback to optimize performance by memoizing values and functions.

Debugging

Utilize useDebugValue to add labels for custom hooks in React DevTools.

Stay Updated

React is constantly evolving, so keep an eye on the latest updates and best practices.

LIKE

SHARE

SAVE

THANK YOU!

follow for more!