React Hooks

NGUYEN TRONG TIEN

useState

`useState` is a React Hook that lets you add a state variable to your component.

The `useState` hook takes a single argument, our initial state, and returns an array containing two elements:

- `state` - the current state
- `setState` - a function to update our state

```
const [state, setState] = useState(initialState);
```

useState

How useState Works

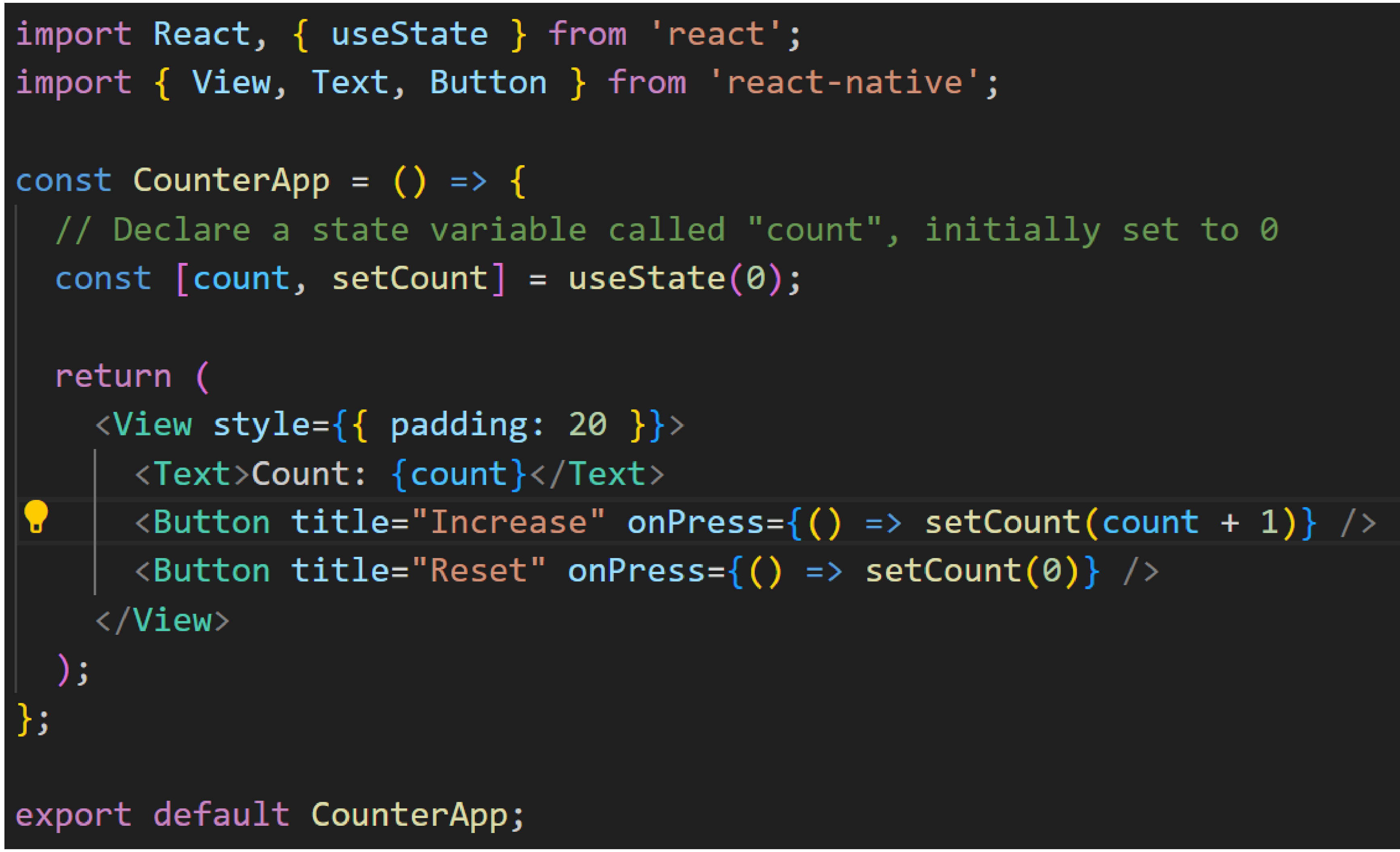
Initialization: When you call **useState**, you provide an initial value for your state variable.

State Value: useState returns two things:

- **The current state value.**
- **A function to update the state.**

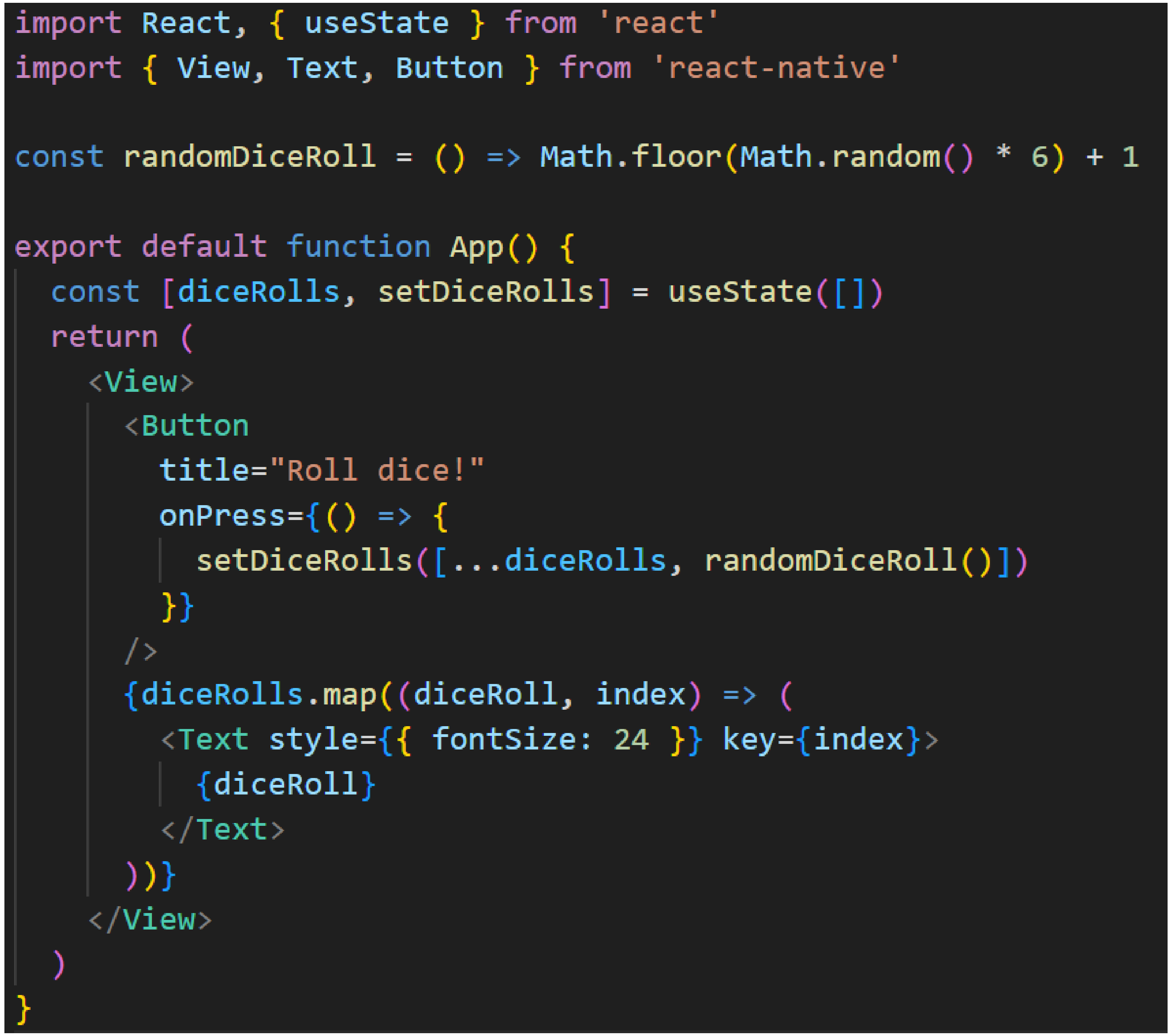
Re-rendering: When the state is updated using the provided function, the component re-renders to reflect the new state.

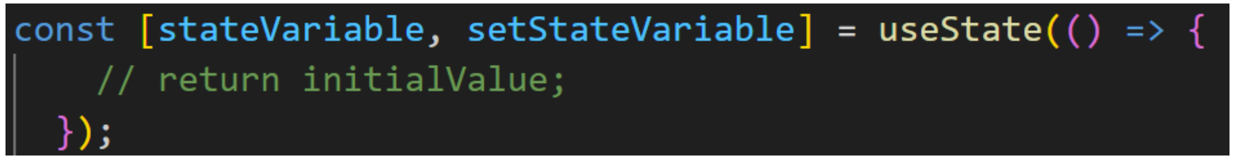
useState



```jsx
import React, { useState } from 'react';
import { View, Text, Button } from 'react-native';

const CounterApp = () => {
  // Declare a state variable called "count", initially set to 0
  const [count, setCount] = useState(0);

  return (
    <View style={{ padding: 20 }}>
      <Text>Count: {count}</Text>
      <Button title="Increase" onPress={() => setCount(count + 1)} />
      <Button title="Reset" onPress={() => setCount(0)} />
    </View>
  );
};

export default CounterApp;
```

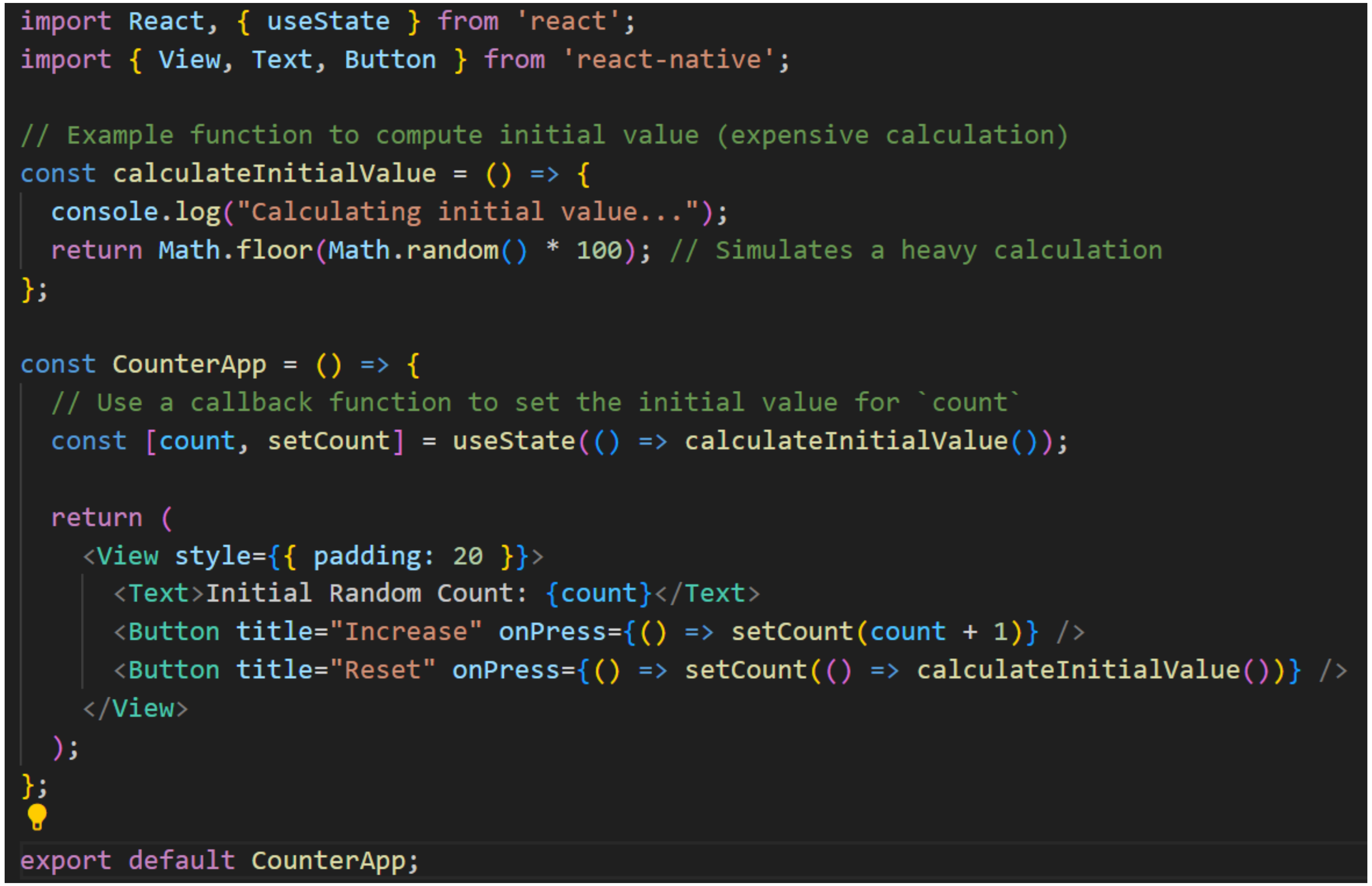
useState


```
import React, { useState } from 'react'
import { View, Text, Button } from 'react-native'

const randomDiceRoll = () => Math.floor(Math.random() * 6) + 1

export default function App() {
  const [diceRolls, setDiceRolls] = useState([])
  return (
    <View>
      <Button
        title="Roll dice!"
        onPress={() => {
          setDiceRolls([...diceRolls, randomDiceRoll()])
        }}
      />
      {diceRolls.map((diceRoll, index) => (
        <Text style={{ fontSize: 24 }} key={index}>
          {diceRoll}
        </Text>
      ))}
    </View>
  )
}
```



```
const [stateVariable, setStateVariable] = useState(() => {
    // return initialValue;
  });
```

useState



```
import React, { useState } from 'react';
import { View, Text, Button } from 'react-native';

// Example function to compute initial value (expensive calculation)
const calculateInitialValue = () => {
  console.log("Calculating initial value...");
  return Math.floor(Math.random() * 100); // Simulates a heavy calculation
};

const CounterApp = () => {
  // Use a callback function to set the initial value for `count`
  const [count, setCount] = useState(() => calculateInitialValue());

  return (
    <View style={{ padding: 20 }}>
      <Text>Initial Random Count: {count}</Text>
      <Button title="Increase" onPress={() => setCount(count + 1)} />
      <Button title="Reset" onPress={() => setCount(() => calculateInitialValue())} />
    </View>
  );
};

export default CounterApp;
```

initialValue: callback function

useState

In React Native (and React), when you pass a callback function to useState, it's typically used when the initial state requires some heavy computation or when you only want to run the function once (on the first render) to calculate the initial value.

This is useful for optimizing performance by preventing the function from being executed on every render.

Use Case

When you need to compute an initial value only once and avoid re-running the calculation on every render.

For example, if you need to fetch data from an API or calculate a large value, using a callback ensures it only happens when necessary (during initial rendering).

This approach is helpful for performance optimization, especially in scenarios where the initial state requires some heavy computation.

useEffect

The `useEffect` hook takes 2 arguments:

- `callback` - a function with side effects
- `dependencies` - an optional array containing dependency values

When our component function runs, the `callback` will be called if any `dependencies` have changed since the last time the component function ran.

useEffect

useEffect(callback)

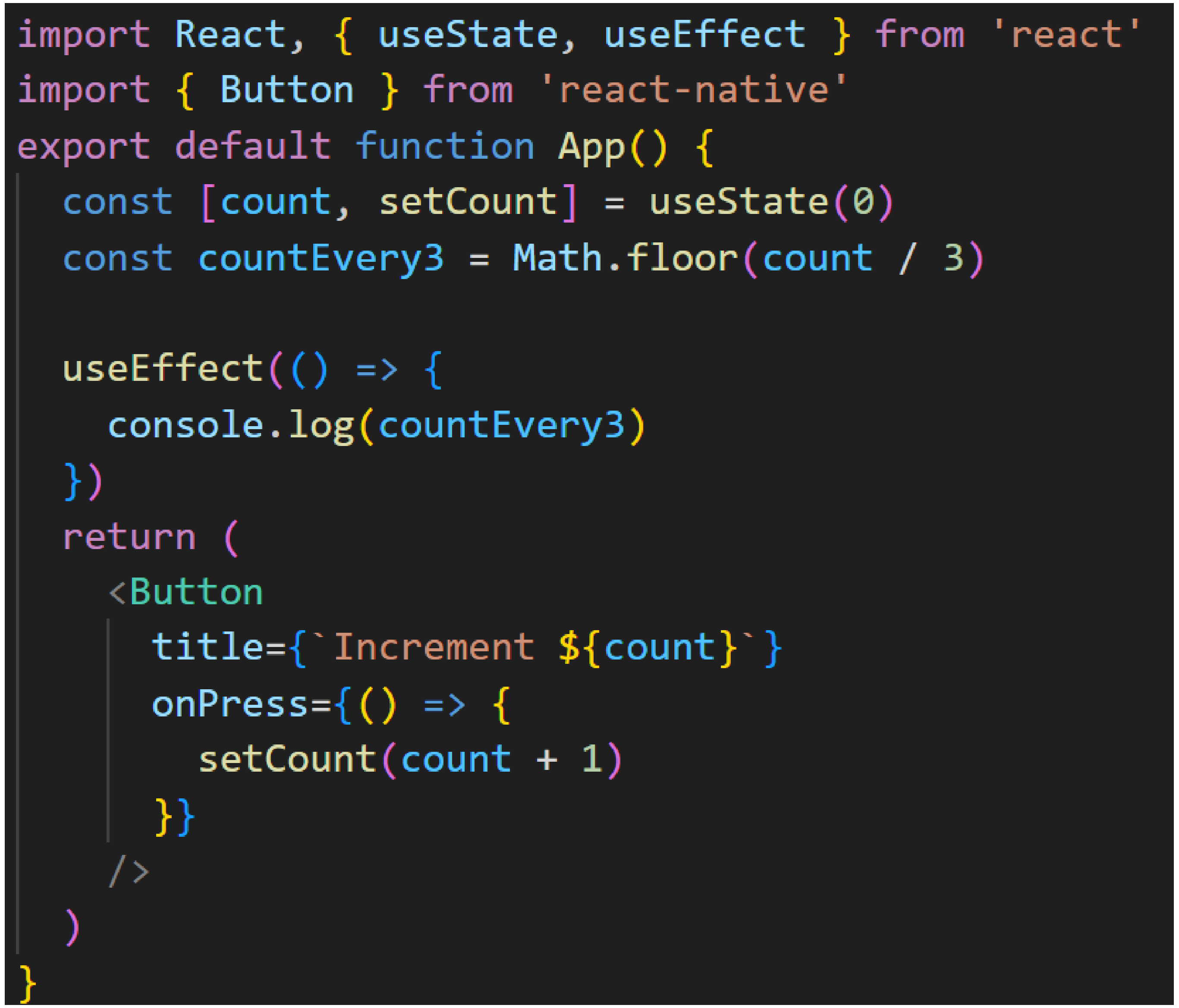
useEffect(callback, [])

useEffect(callback, [deps])

callback call after **Component finish render**

useEffect(callback)

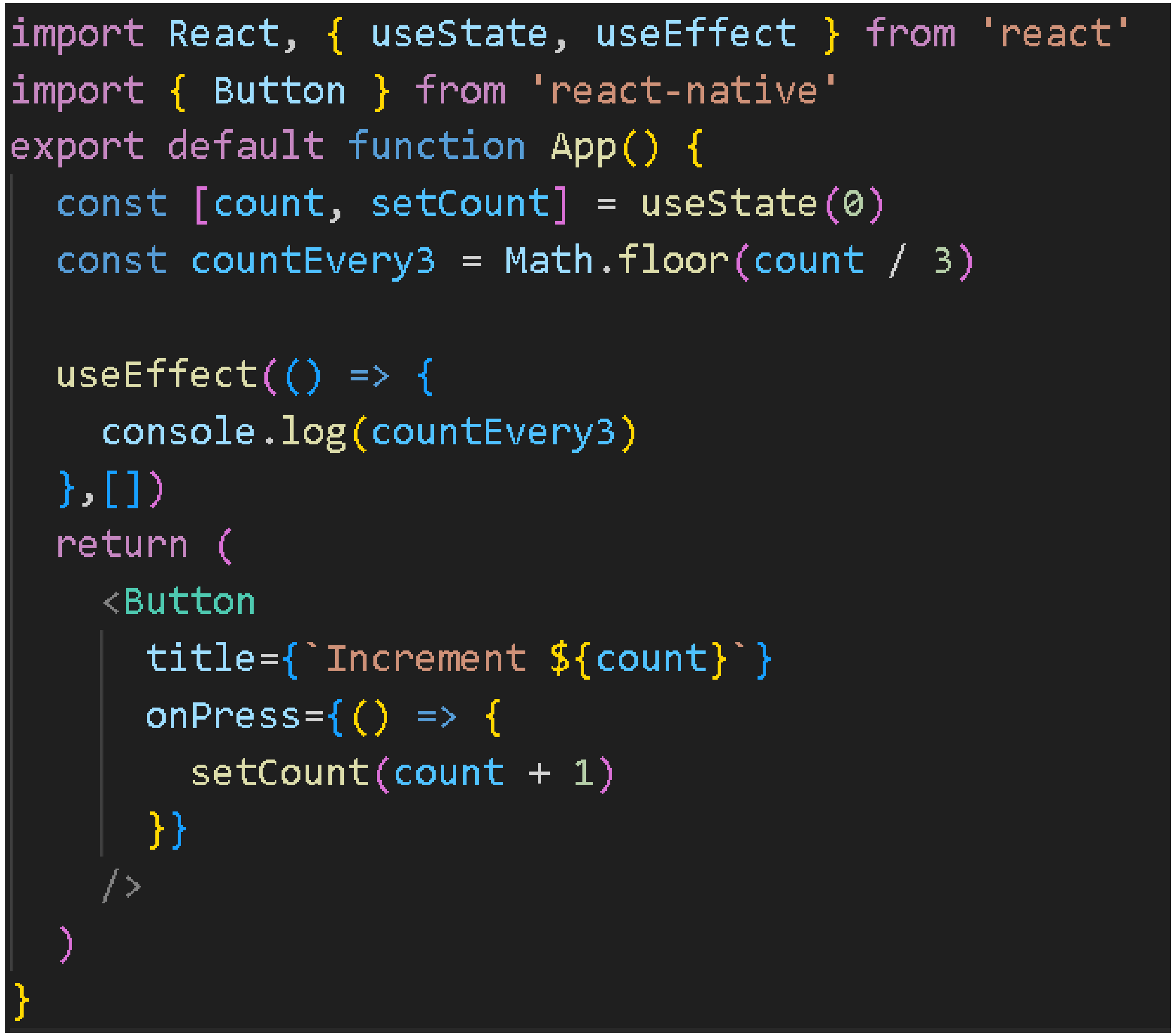
callback call after Component finish render



```
import React, { useState, useEffect } from 'react'
import { Button } from 'react-native'
export default function App() {
  const [count, setCount] = useState(0)
  const countEvery3 = Math.floor(count / 3)

  useEffect(() => {
    console.log(countEvery3)
  })
  return (
    <Button
      title={`Increment ${count}`}
      onPress={() => {
        setCount(count + 1)
      }}
    />
  )
}
```

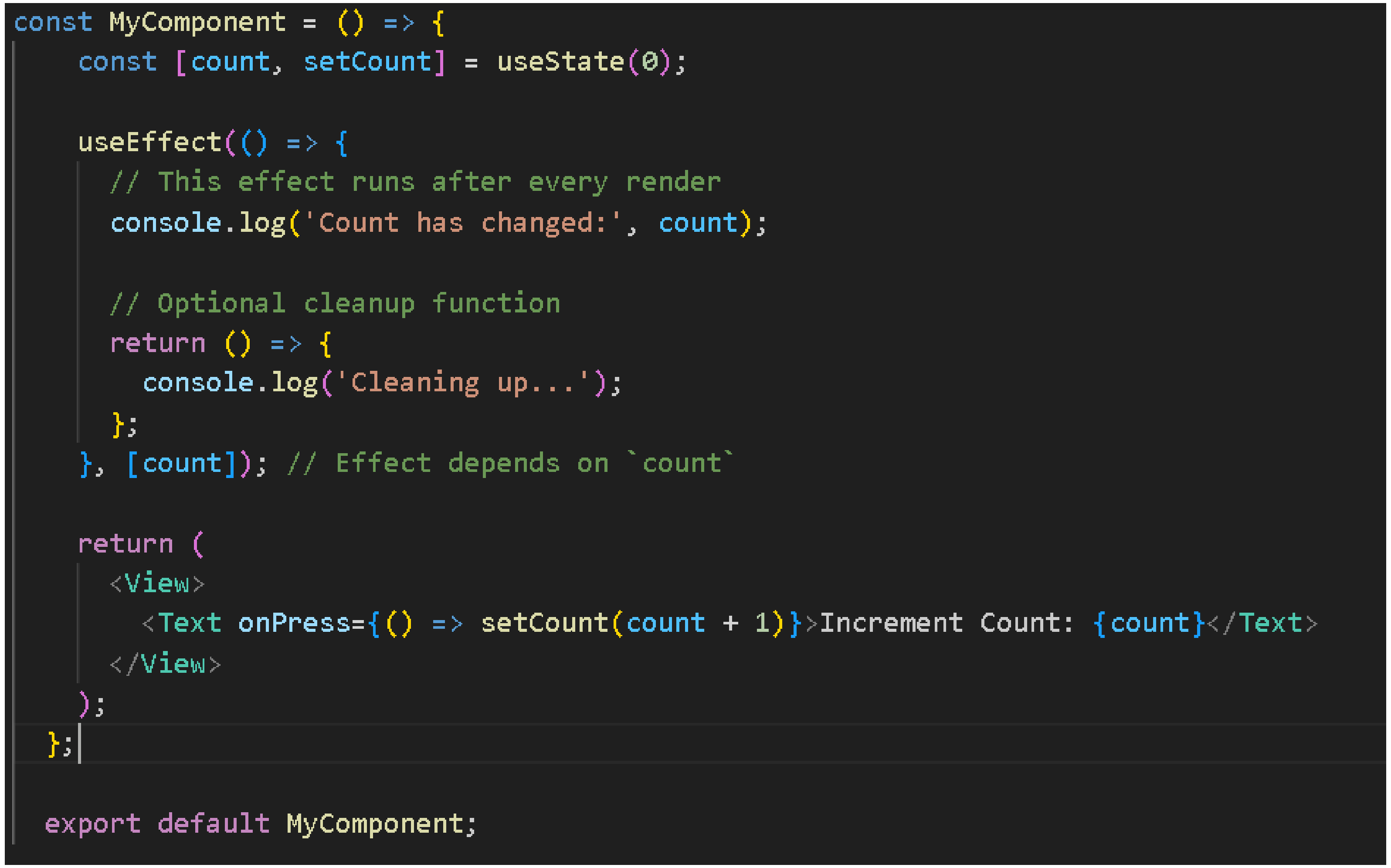
useEffect(callback, [])

callback call **only 1 time** after Component finish render



```
import React, { useState, useEffect } from 'react'
import { Button } from 'react-native'
export default function App() {
  const [count, setCount] = useState(0)
  const countEvery3 = Math.floor(count / 3)

  useEffect(() => {
    console.log(countEvery3)
  },[])
  return (
    <Button
      title={`Increment ${count}`}
      onPress={() => {
        setCount(count + 1)
      }}
    />
  )
}
```

useEffect(callback, [deps])



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

    useEffect(() => {
      // This effect runs after every render
      console.log('Count has changed:', count);

      // Optional cleanup function
      return () => {
        console.log('Cleaning up...');
      };
    }, [count]); // Effect depends on `count`

    return (
      <View>
        <Text onPress={() => setCount(count + 1)}>Increment Count: {count}</Text>
      </View>
    );
  };

  export default MyComponent;
```

useEffect–explanation

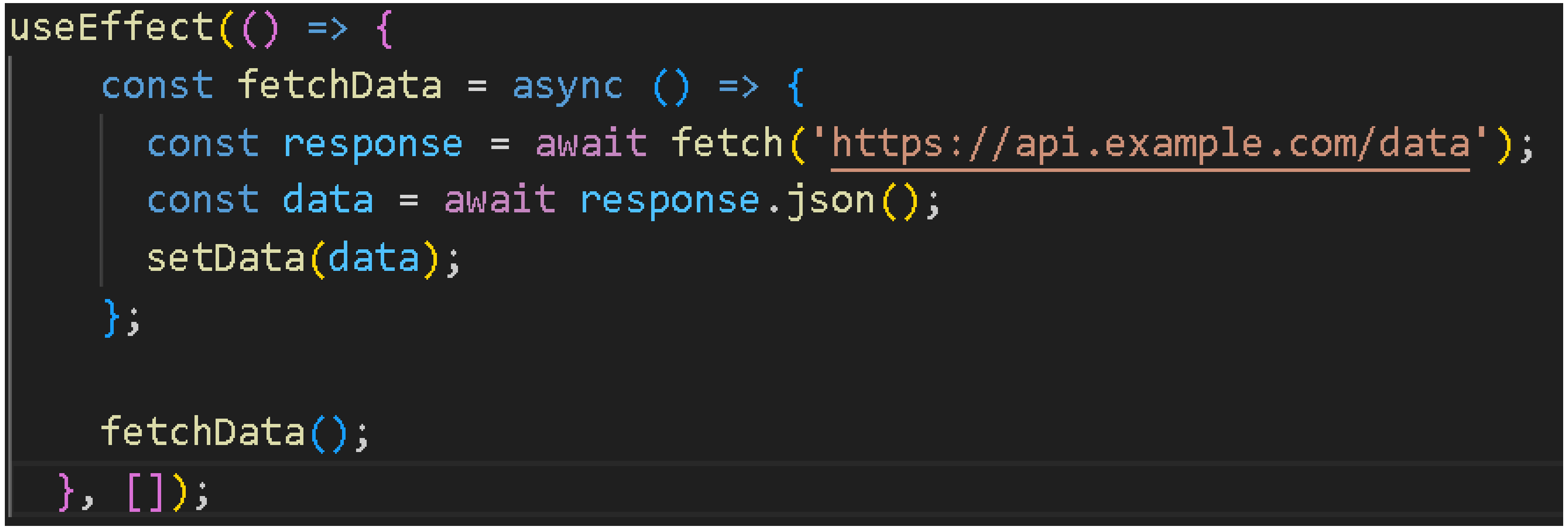
Effect Function: The function you provide will run after the component renders. In this example, it logs the count value whenever it changes.

Cleanup Function: If your effect creates subscriptions or timers, you can return a cleanup function. This function will run before the component unmounts or before the effect runs again.

Dependency Array: The effect only runs when the specified dependencies change. If you pass an empty array ([]), the effect runs only once after the initial render, similar to ***componentDidMount***.

useEffect

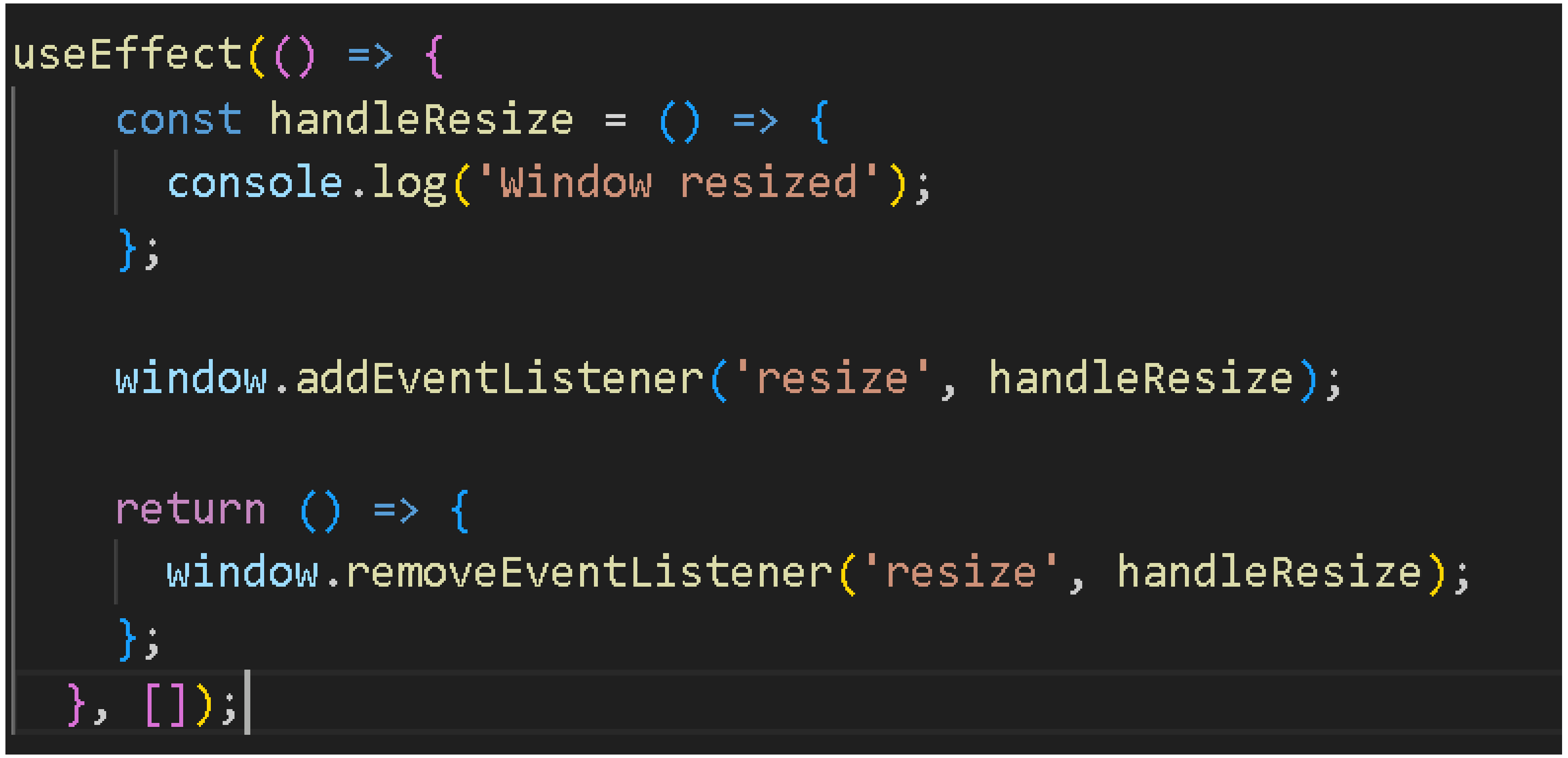
Fetching Data



```
useEffect(() => {
    const fetchData = async () => {
      const response = await fetch('https://api.example.com/data');
      const data = await response.json();
      setData(data);
    };

    fetchData();
  }, []);
```

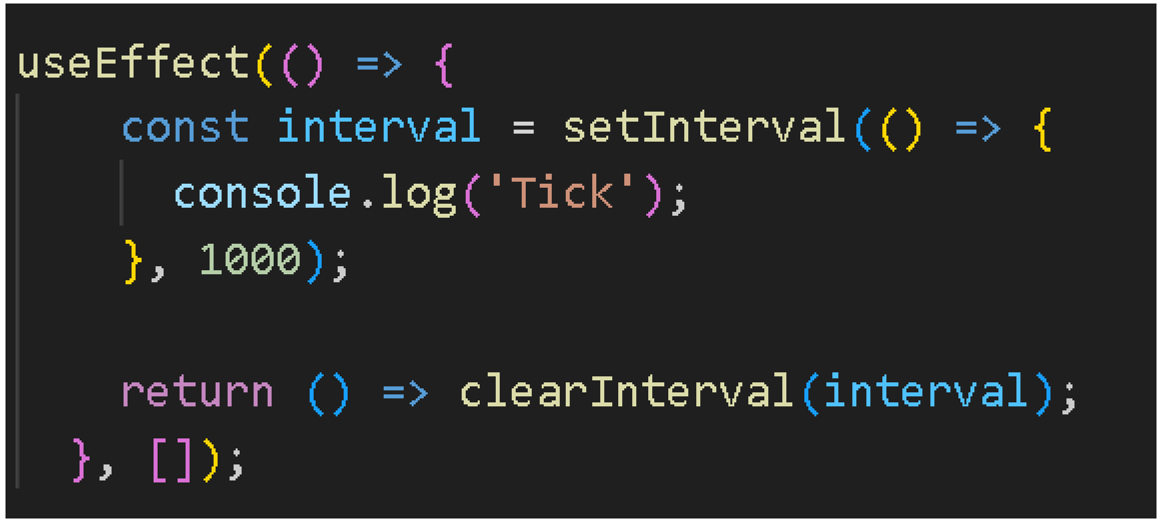
useEffect

Event Listeners


```
useEffect(() => {
    const handleResize = () => {
      console.log('Window resized');
    };

    window.addEventListener('resize', handleResize);

    return () => {
      window.removeEventListener('resize', handleResize);
    };
  }, []);
```

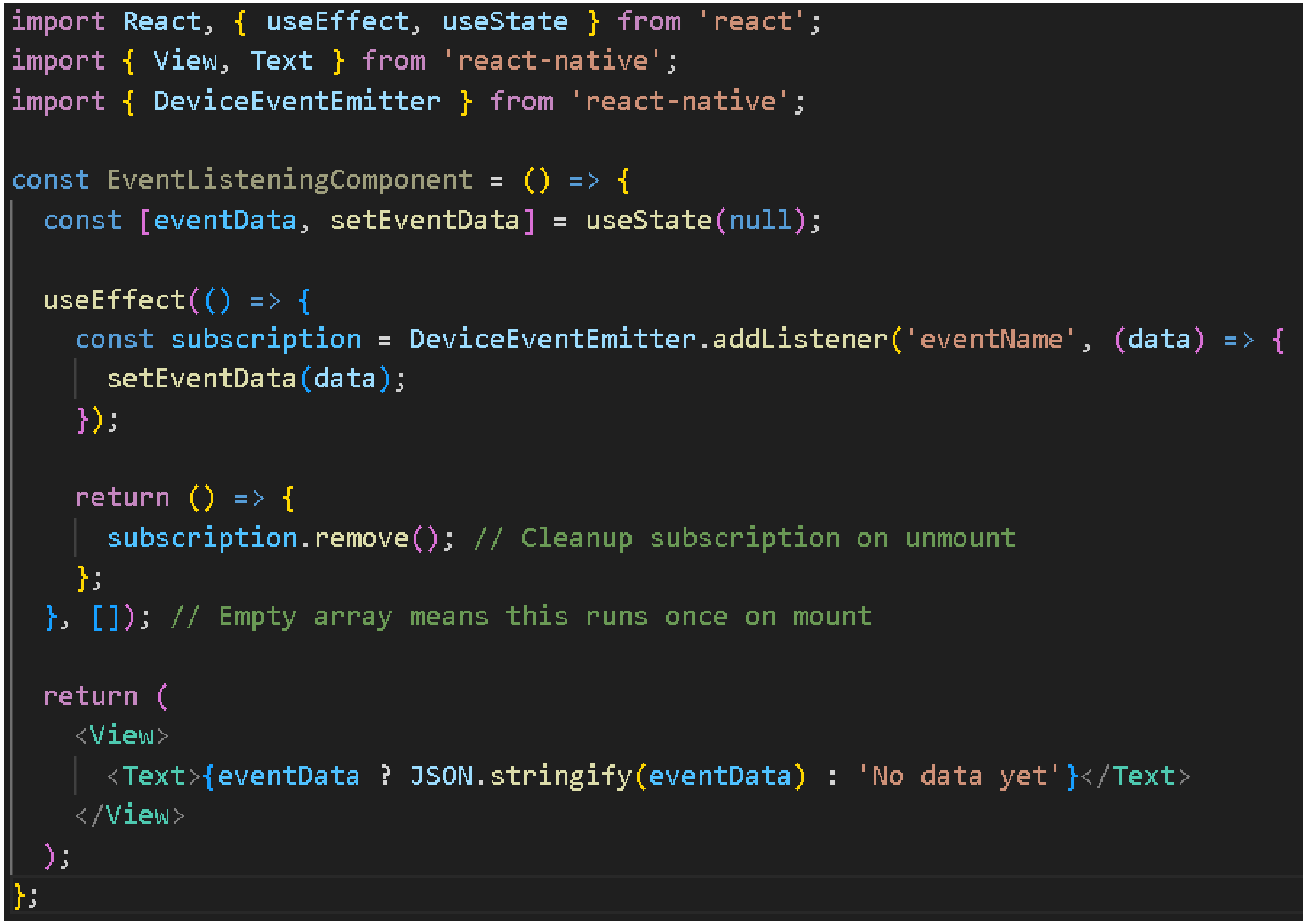
useEffect

Animation and Timers



```
useEffect(() => {
    const interval = setInterval(() => {
      console.log('Tick');
    }, 1000);

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

useEffect



```
import React, { useEffect, useState } from 'react';
import { View, Text } from 'react-native';
import { DeviceEventEmitter } from 'react-native';

const EventListeningComponent = () => {
  const [eventData, setEventData] = useState(null);

  useEffect(() => {
    const subscription = DeviceEventEmitter.addListener('eventName', (data) => {
      setEventData(data);
    });

    return () => {
      subscription.remove(); // Cleanup subscription on unmount
    };
  }, []); // Empty array means this runs once on mount

  return (
    <View>
      <Text>{eventData ? JSON.stringify(eventData) : 'No data yet'}</Text>
    </View>
  );
};
```

useEffect vs Class Component

NGUYEN TRONG TIEN

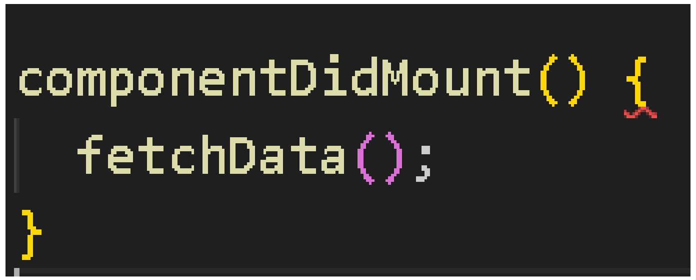
Class Component

- Let's break down the differences and similarities between **useEffect** in functional components and the lifecycle methods in class components:
  - **componentDidMount**
  - **componentDidUpdate**
  - **componentWillUnmount**.

componentDidMount

When It Runs: After the component has been rendered for the first time.

UseCase: Ideal for fetching data, setting up subscriptions, or manipulating the DOM.

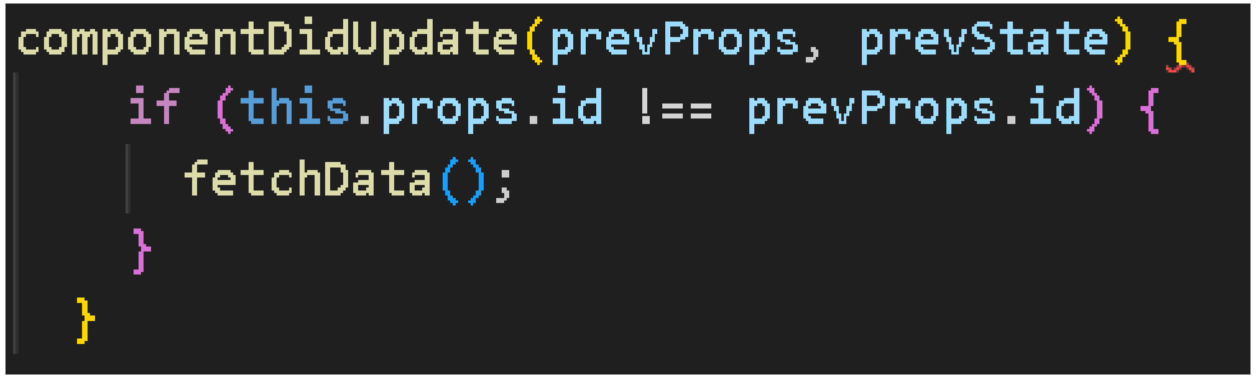


```
componentDidMount() {
  fetchData();
}
```

componentDidUpdate

When It Runs: After the component updates (i.e., re-renders due to state or prop changes)**.**

Use Case: Good for responding to prop changes or state updates.

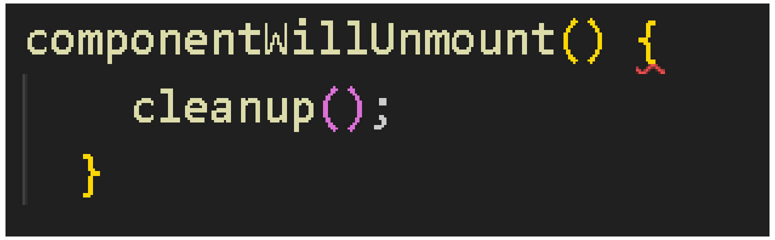


```
componentDidUpdate(prevProps, prevState) {
    if (this.props.id !== prevProps.id) {
      fetchData();
    }
  }
```

componentWillUnmount

When It Runs: Right before the component is removed from the DOM.

Use Case: Useful for cleaning up subscriptions or timers.



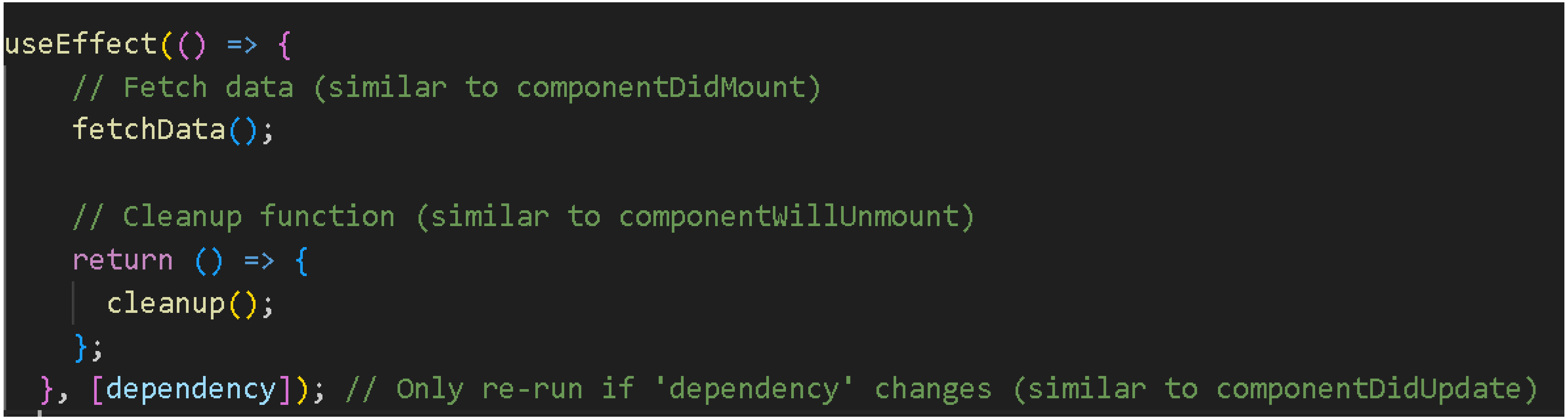
```
componentWillUnmount() {
    cleanup();
  }
```

Functional Components with useEffect

When It Runs: Runs after the render, similar to componentDidMount and componentDidUpdate. It can also be configured to run only on mount, only on updates, or only on unmount.

Use Case: Covers all use cases for the three lifecycle methods in one API.

Example



```
useEffect(() => {
    // Fetch data (similar to componentDidMount)
    fetchData();

    // Cleanup function (similar to componentWillUnmount)
    return () => {
      cleanup();
    };
  }, [dependency]); // Only re-run if 'dependency' changes (similar to componentDidUpdate)
```

Key Differences

Conciseness: **useEffect** consolidates multiple lifecycle methods into a single hook, making it easier to manage effects.

Dependencies: With **useEffect**, you specify dependencies that determine when the effect should re-run, allowing for more granular control.

Cleanup: The cleanup logic is returned from the effect function, streamlining the process of handling side effects.

Less Boilerplate: Functional components are generally simpler and require less boilerplate code compared to class components.

What is useRef?

With ***useRef*** we can create and update a single mutable value that exists for the lifetime of the component instance.

After assigning the ref to a variable, we use .current to access the mutable value.

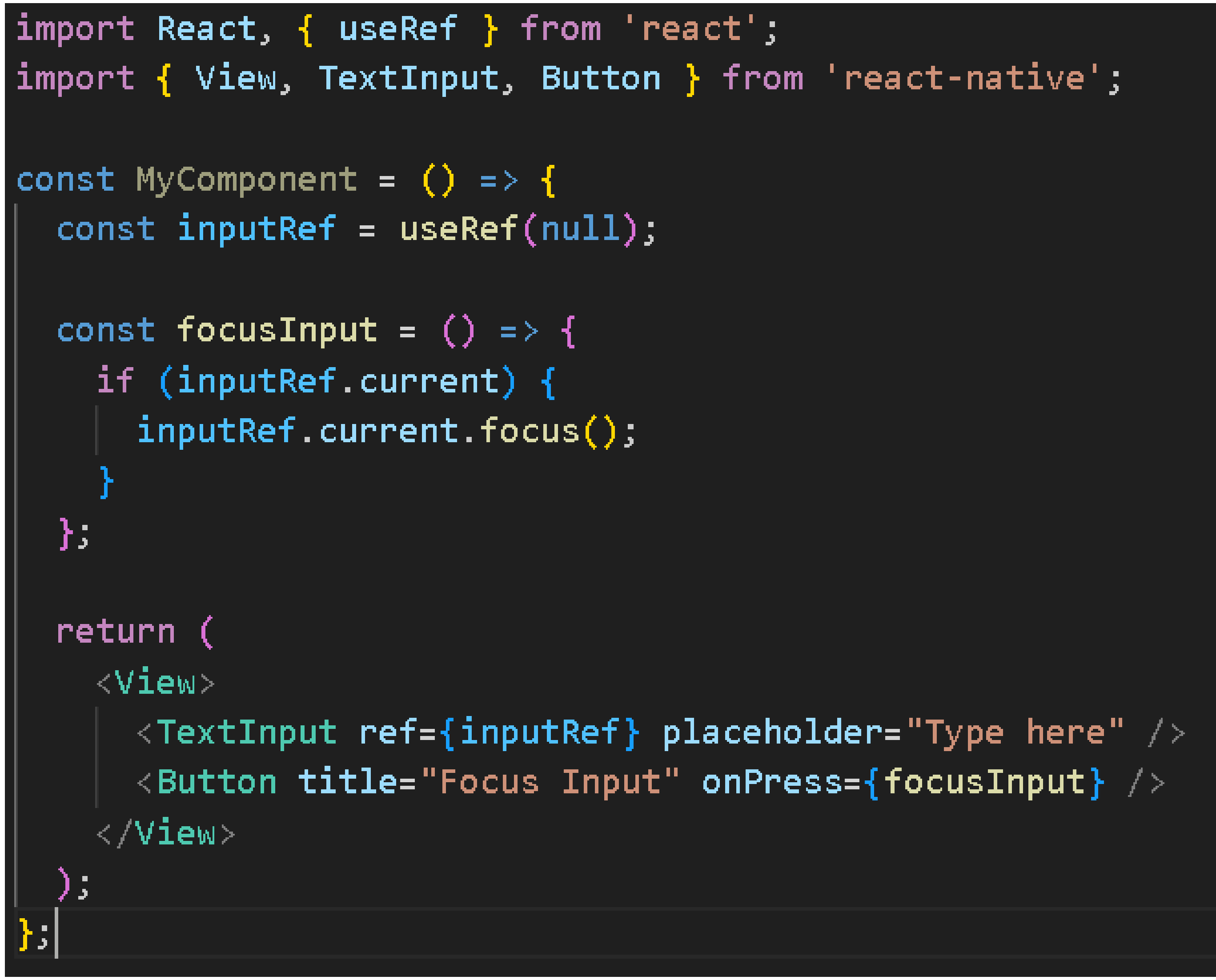
It can be used to hold a reference to a DOM element, or any mutable value that you want to keep around without causing a re-render when updated.

Common Use Cases

Accessing DOM Elements:

You can use useRef to directly reference a component or a DOM element in your React Native application.

This is useful for focus management, measuring elements, or integrating with third-party libraries.

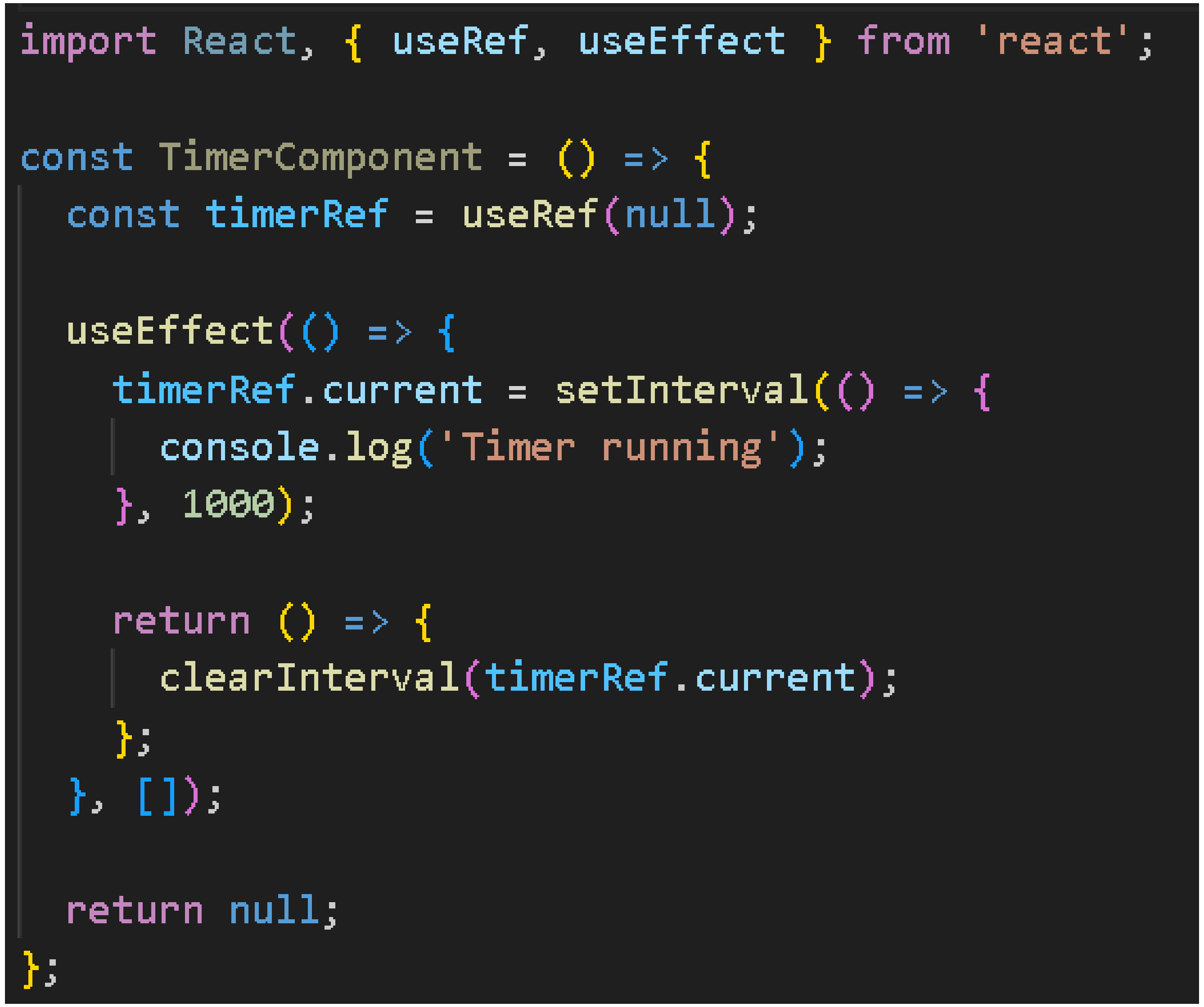


```
import React, { useRef } from 'react';
import { View, TextInput, Button } from 'react-native';

const MyComponent = () => {
  const inputRef = useRef(null);

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

  return (
    <View>
      <TextInput ref={inputRef} placeholder="Type here" />
      <Button title="Focus Input" onPress={focusInput} />
    </View>
  );
};
```

Common Use Cases

Storing Mutable Values:

You can store any mutable value (like timers or intervals) that doesn’t require re-rendering the component when updated.



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

const TimerComponent = () => {
  const timerRef = useRef(null);

  useEffect(() => {
    timerRef.current = setInterval(() => {
      console.log('Timer running');
    }, 1000);

    return () => {
      clearInterval(timerRef.current);
    };
  }, []);

  return null;
};
```

Common Use Cases

Avoiding Re-renders:

Since updating the .current property of a useRef does not trigger a re-render, it's useful for holding values that you want to keep updated without affecting the rendering cycle.

useRef

```
import React, { useState, useEffect, useRef } from 'react'
import { View, Text, Button } from 'react-native'

export default function App() {
  const intervalRef = useRef()
  const [count, setCount] = useState(0)

  useEffect(() => {
    intervalRef.current = setInterval(  () => setCount((count) => count + 1),  1000    )
    return () => {     clearInterval(intervalRef.current)    }  }, [])

  return (
    <View>
      <Text style={{ fontSize: 120 }}>{count}</Text>
      <Button
        title="Stop"
        onPress={() => {
          clearInterval(intervalRef.current)
        }}
      />
    </View>
  )
}
```

Differences from Other Hooks

Unlike useState: Changing a state variable causes a component re-render. In contrast, changing a ref does not trigger a re-render, which can be advantageous for performance in certain situations.

Unlike useEffect: *useRef* does not execute effects when its value changes; it's a way to hold references or values between renders without side effects.

useContext

The ***useContext*** hook in React Native (and React in general) provides a way to access context values in a functional component.

Context is a feature that allows you to share values (***like themes, user information, or other global data***) across your component tree without having to pass props down manually at every level.

What is Context?

Context API: The **Context API** in React allows you to create global data that can be accessed by any component in the component tree, regardless of how deep it is nested.

Provider and Consumer: You create a context with **React.createContext()**, and you provide values using a Provider component. Components that need access to these values can consume them using useContext.

Using useContext

Creating a Context

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

const MyContext = createContext();
```

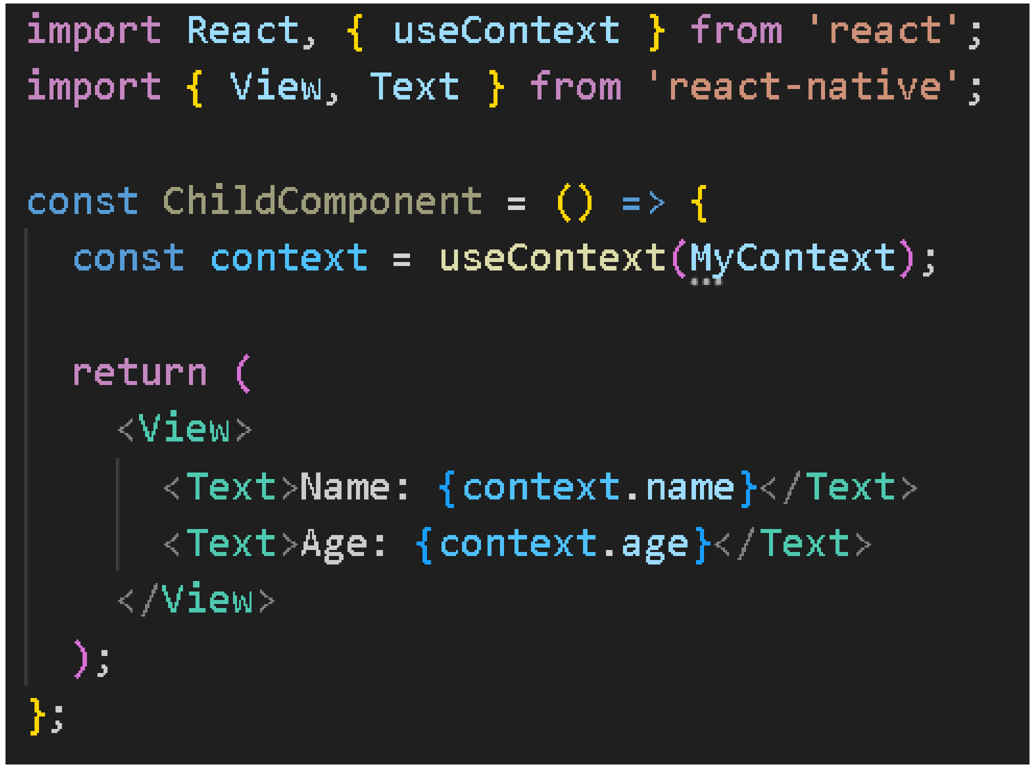
Providing Context Values

```
const App = () => {
    const value = { name: 'John Doe', age: 30 };

    return (
      <MyContext.Provider value={value}>
        <ChildComponent />
      </MyContext.Provider>
    );
  };
```

Using useContext

Consuming Context Values with

useContext



```
import React, { useContext } from 'react';
import { View, Text } from 'react-native';

const ChildComponent = () => {
  const context = useContext(MyContext);

  return (
    <View>
      <Text>Name: {context.name}</Text>
      <Text>Age: {context.age}</Text>
    </View>
  );
};
```

Benefits of useContext

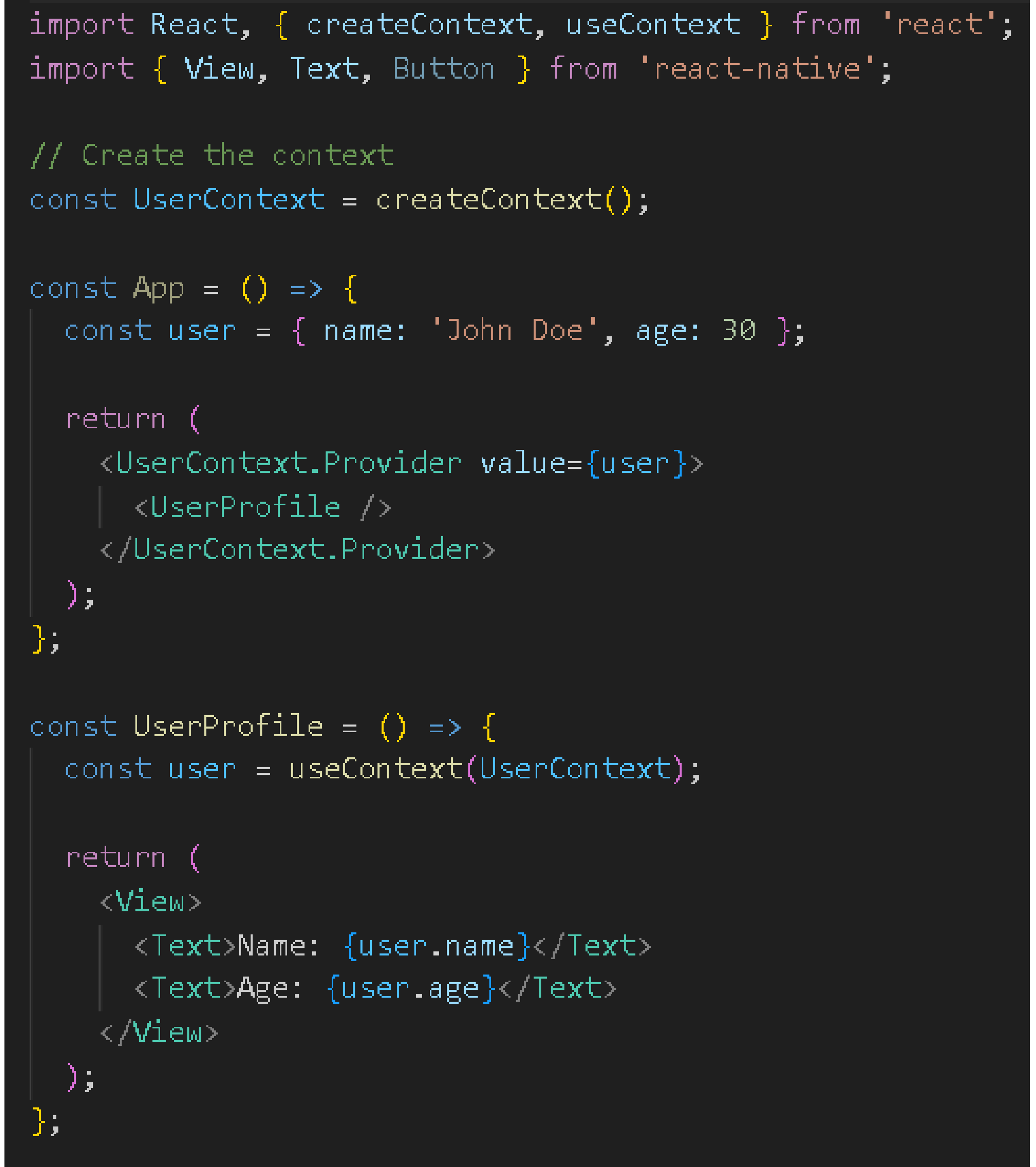
Cleaner Code: It avoids "prop drilling," where you have to pass props through many layers of components. Instead, you can access the data directly where it's needed.

Simplified State Management: useContext is often used alongside useReducer for state management, providing a way to manage complex state in a more organized way.

Dynamic Updates: When the context value changes, all components that consume the context will automatically re-render with the new value.

Example of Full Usage

The **useContext** hook is a powerful feature in RN that simplifies state management and allows for easy access to global data without prop drilling. It works seamlessly with the Context API to create a more maintainable and scalable application structure.



```
import React, { createContext, useContext } from 'react';
import { View, Text, Button } from 'react-native';

// Create the context
const UserContext = createContext();

const App = () => {
  const user = { name: 'John Doe', age: 30 };

  return (
    <UserContext.Provider value={user}>
      <UserProfile />
    </UserContext.Provider>
  );
};

const UserProfile = () => {
  const user = useContext(UserContext);

  return (
    <View>
      <Text>Name: {user.name}</Text>
      <Text>Age: {user.age}</Text>
    </View>
  );
};
```

useReducer

The useReducer hook in RN (and React in general) is a powerful tool for managing complex state in functional components.

It’s particularly useful when you have multiple state variables that depend on each other or when you need to handle complex state transitions, similar to how you would with **Redux**.

useReducer

The useReducer hook requires 2 arguments, and has an optional 3rd argument:

- reducer - a pure function that takes a state and an action, and returns a new state value based on the action
- initialState - any initial state value, just like useState
- initializer (optional) - this is uncommon, but we'll briefly introduce it later.

The useReducer hook returns the current state, and a dispatch function to update the state.

What is useReducer?

State Management: **useReducer** provides a way to manage state with a reducer function, similar to how **Redux** manages state. It allows you to **update state based on actions**, making it easier to handle complex updates in a predictable manner.

Basic Usage of useReducer

*The **useReducer** hook takes two arguments:*

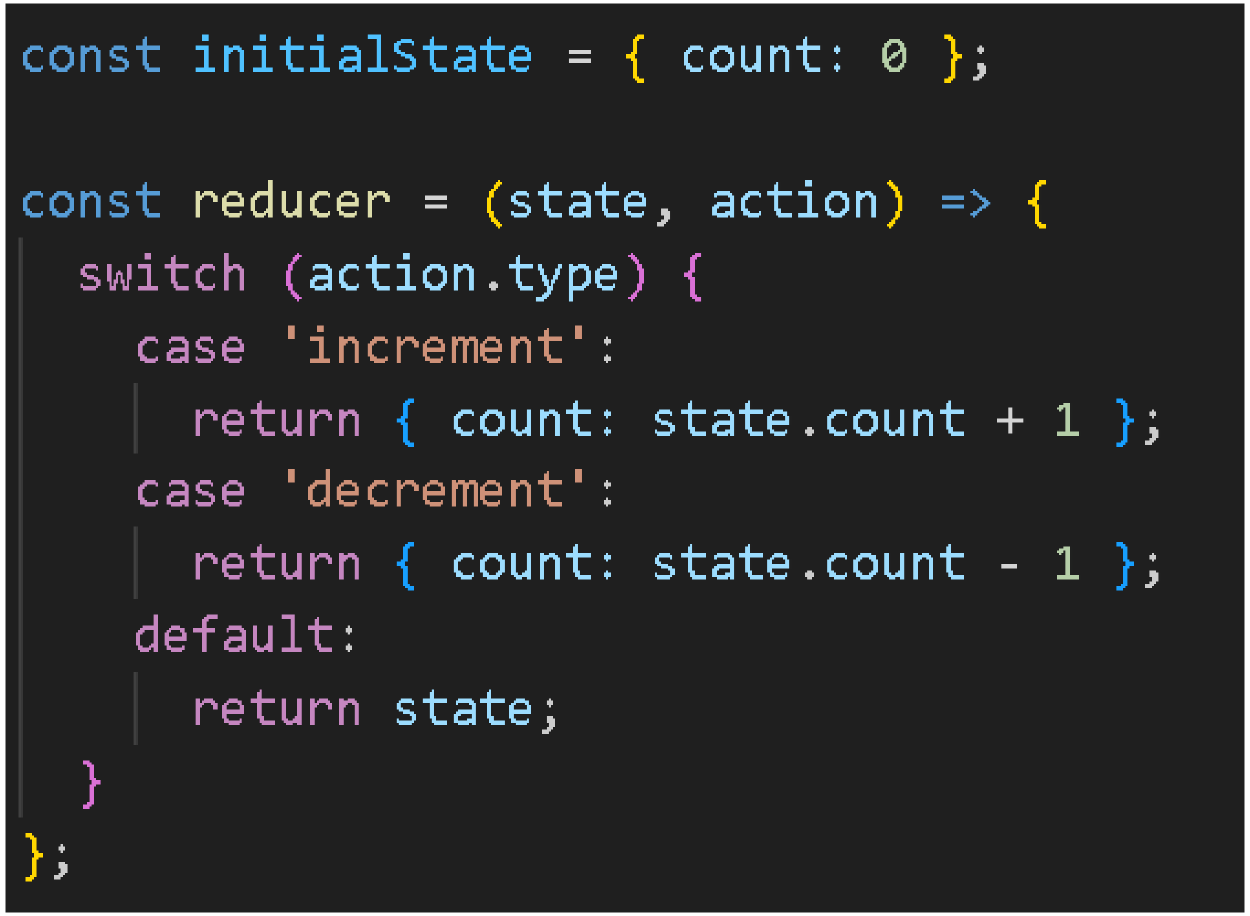
1. A reducer function that defines how the state updates in response to actions.
2. An initial state.

It returns the **current state** and a **dispatch function** to **send actions to the reducer.**

Step-by-Step Example

1. **Define the Reducer Function:**

   The reducer function takes two parameters: the current state and an action. Based on the action type, it returns a new state.

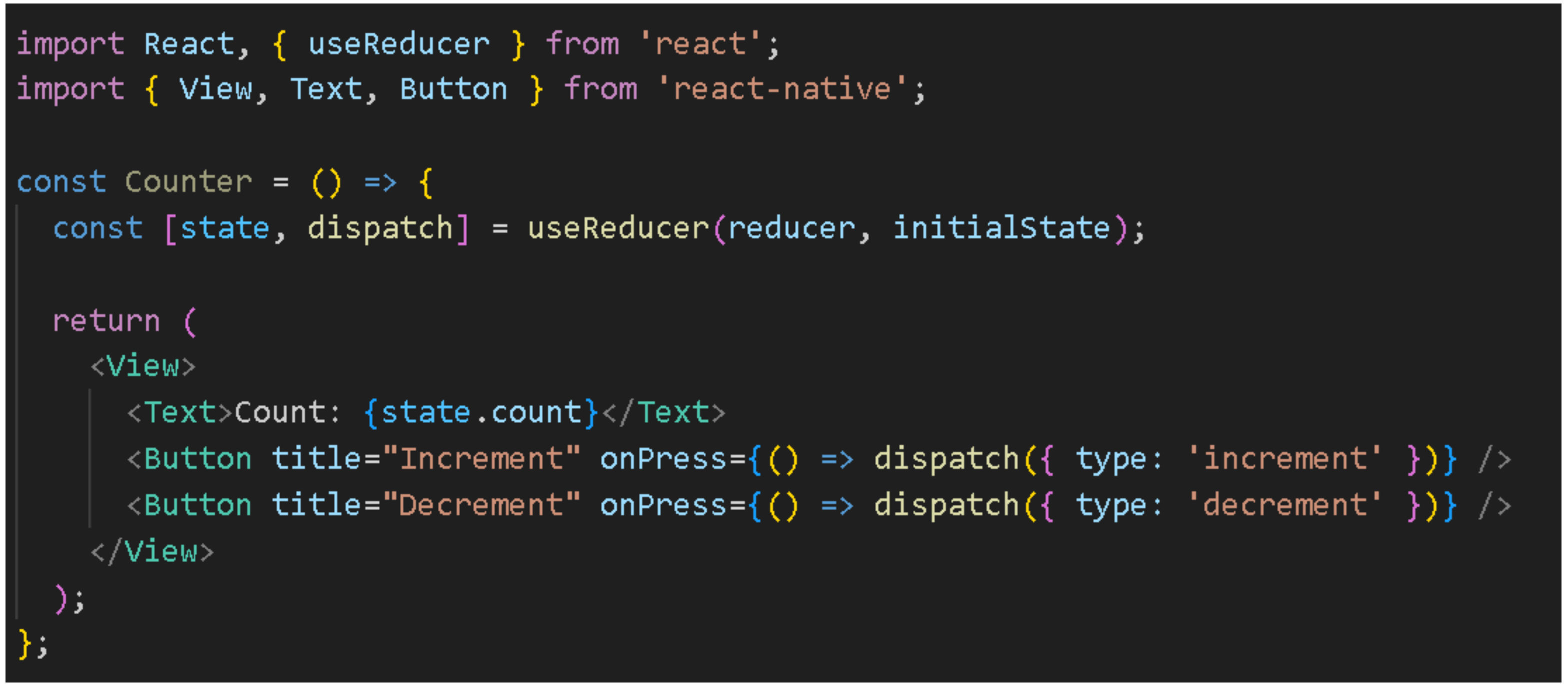


```javascript
const initialState = { count: 0 };

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

Step-by-Step Example

2. Use useReducer in Your Component

Initialize state with useReducer and use the dispatch function to update the state.

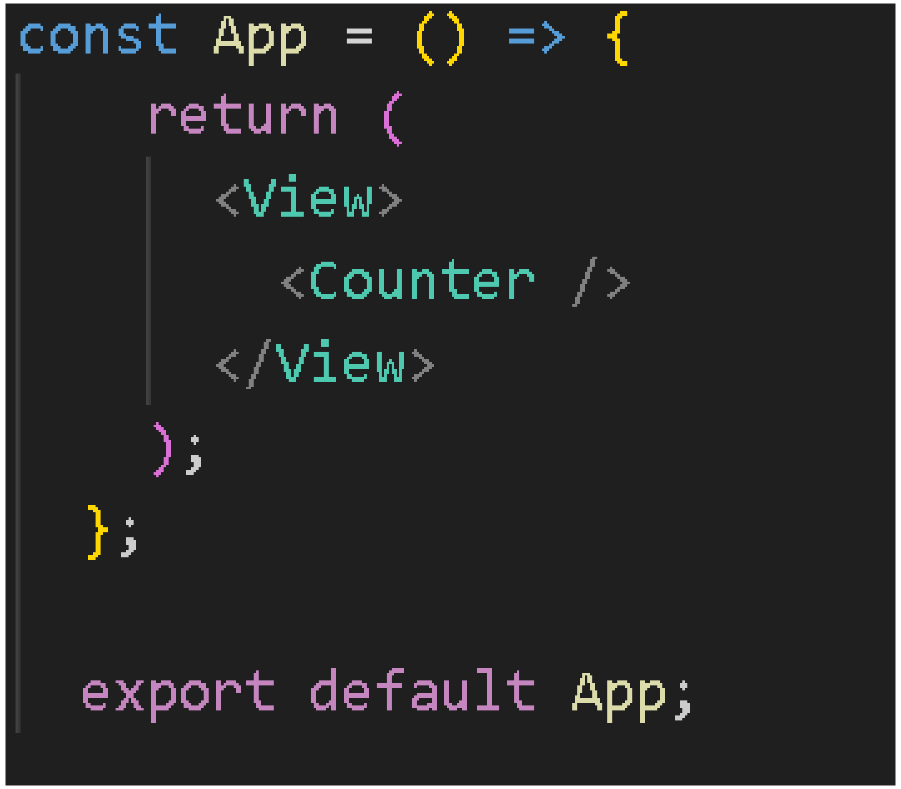


```
import React, { useReducer } from 'react';
import { View, Text, Button } from 'react-native';

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

  return (
    <View>
      <Text>Count: {state.count}</Text>
      <Button title="Increment" onPress={() => dispatch({ type: 'increment' })} />
      <Button title="Decrement" onPress={() => dispatch({ type: 'decrement' })} />
    </View>
  );
};
```

Step-by-Step Example

3. Render Your Component

You can now use the Counter component in your application.



```
const App = () => {
    return (
      <View>
        <Counter />
      </View>
    );
  };

  export default App;
```

Benefits of useReducer

Centralized State Logic: It allows you to centralize state logic in one place, making it easier to understand and manage complex state transitions.

Predictable State Updates: The use of actions and a reducer function makes state updates predictable, similar to how Redux works.

Better for Complex State: It’s ideal for components with complex state logic, such as forms with multiple input fields or components that need to track multiple related values.

When to Use useReducer vs. useState

Use useState: When you have simple state logic or a few state variables that don’t depend on each other.

Use useReducer: When managing more complex state logic, especially if state transitions depend on previous state values or involve multiple sub-values.

Conclusion

The **useReducer** hook in RN is a powerful tool for managing complex state in functional components.

By using a reducer function to handle state updates, you can create more predictable and maintainable state management logic, similar to that of Redux, without the need for additional libraries.

This makes it an excellent choice for applications with intricate state requirements.

What is useMemo?

In React Native, **useMemo** works in exactly the same way as it does in React for web development.

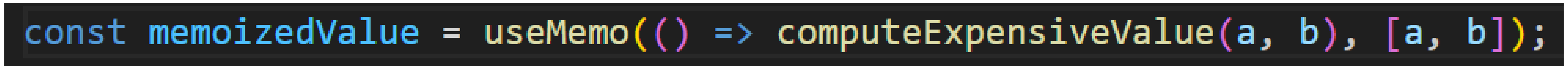
Since React Native shares many concepts with React, including its core hooks, the purpose and usage of useMemo remain consistent.

What is useMemo?

useMemo in RN is a hook used to optimize performance by memoizing the result of an expensive computation so that the result is only recalculated when its dependencies change.

Without memoization, computations inside components may be recalculated unnecessarily on every render, potentially leading to performance bottlenecks, especially on mobile devices where resources are often more constrained than on desktop environments.

Syntax



```
const memoizedValue = useMemo(() => computeExpensiveValue(a, b), [a, b]);
```

computeExpensiveValue(a, b): The function that performs a potentially costly operation.

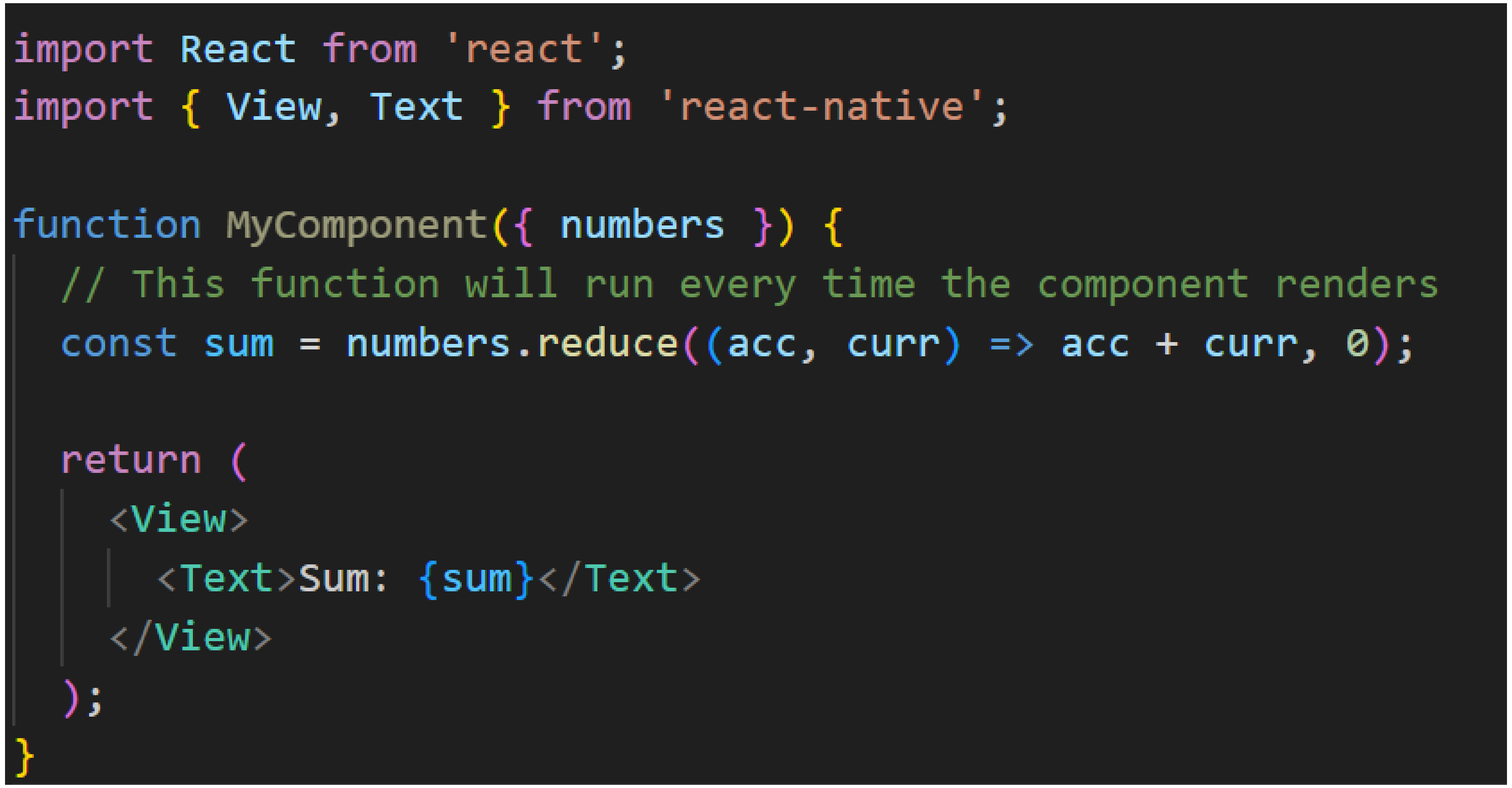
[a, b]: The dependency array. When the values of a or b change, useMemo will recompute the value. If they don’t change, the memoized value is returned without re-executing the function.

Why Use useMemo in React Native?

Mobile devices often **have limited CPU and memory resources**, so unnecessary recalculations can impact app performance.

For example, performing complex calculations, sorting or filtering lists, or computing values in large datasets could lead to poor user experiences. **useMemo** helps to avoid these issues by caching values and recalculating them only when necessary.

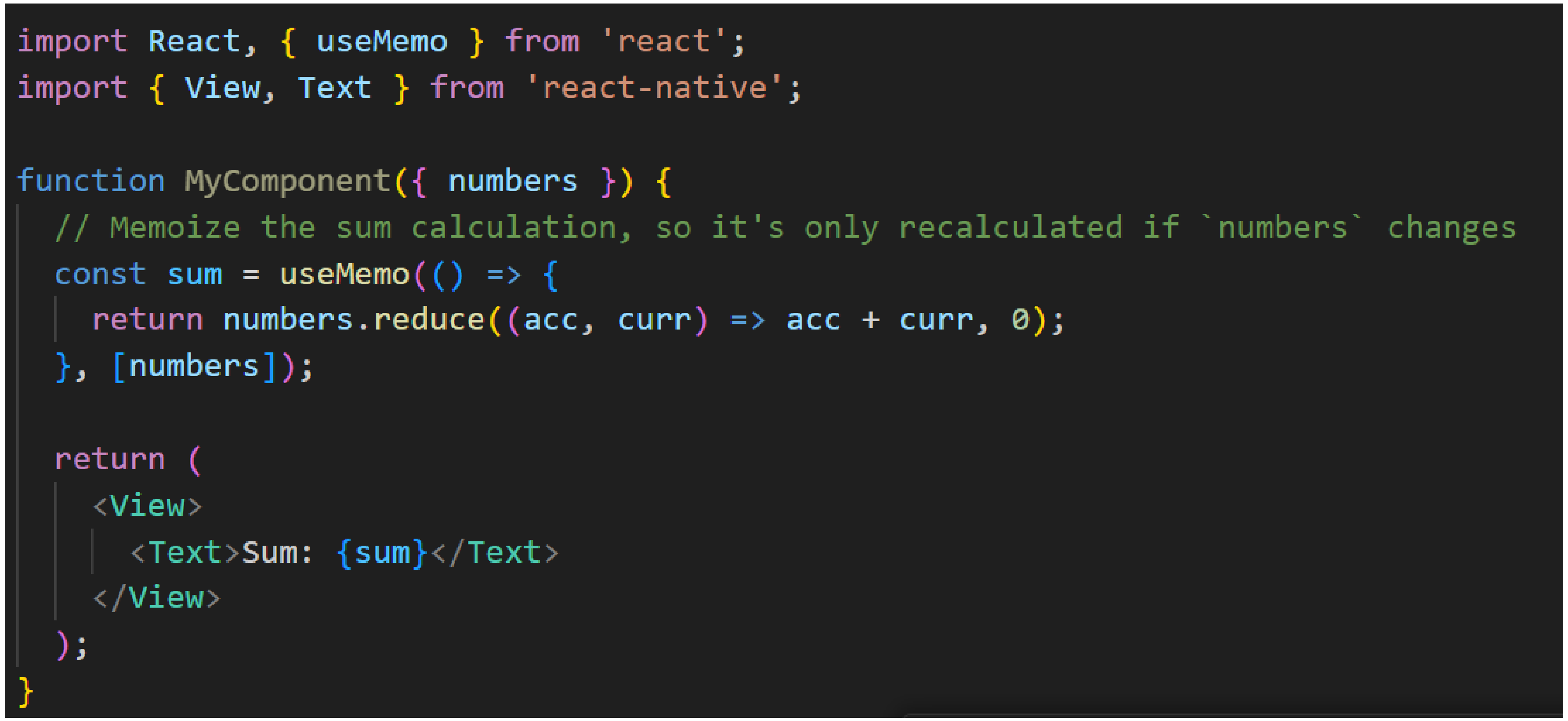
Example in React Native–Without useMemo



```
import React from 'react';
import { View, Text } from 'react-native';

function MyComponent({ numbers }) {
  // This function will run every time the component renders
  const sum = numbers.reduce((acc, curr) => acc + curr, 0);

  return (
    <View>
      <Text>Sum: {sum}</Text>
    </View>
  );
}
```

Here, the sum is recalculated on every render, which can be inefficient for large arrays.

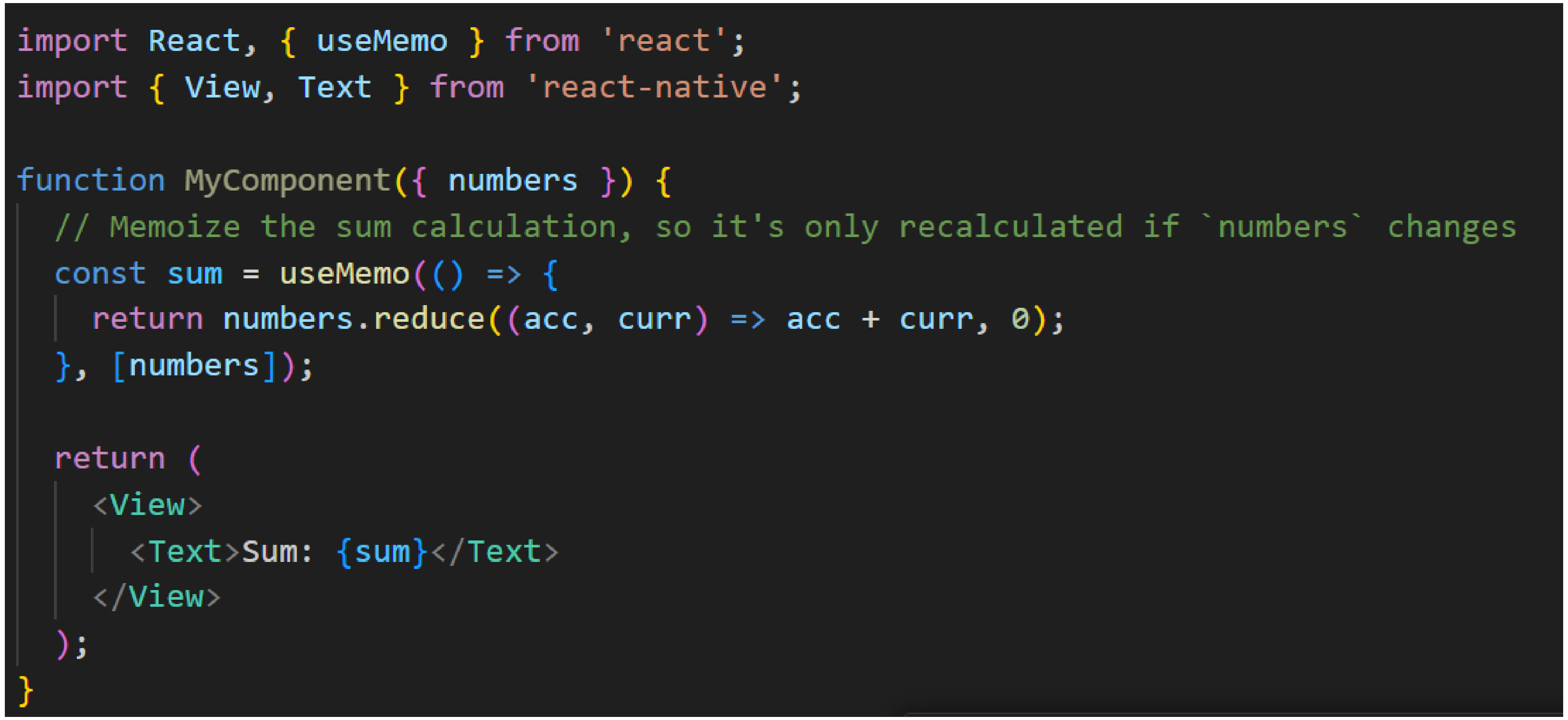
Example in React Native-With useMemo



```
import React, { useMemo } from 'react';
import { View, Text } from 'react-native';

function MyComponent({ numbers }) {
  // Memoize the sum calculation, so it's only recalculated if `numbers` changes
  const sum = useMemo(() => {
    return numbers.reduce((acc, curr) => acc + curr, 0);
  }, [numbers]);

  return (
    <View>
      <Text>Sum: {sum}</Text>
    </View>
  );
}
```

In this example, the sum calculation will only run when the numbers array changes, preventing unnecessary recalculations and improving performance.

Example in React Native-With useMemo



```
import React, { useMemo } from 'react';
import { View, Text } from 'react-native';

function MyComponent({ numbers }) {
  // Memoize the sum calculation, so it's only recalculated if `numbers` changes
  const sum = useMemo(() => {
    return numbers.reduce((acc, curr) => acc + curr, 0);
  }, [numbers]);

  return (
    <View>
      <Text>Sum: {sum}</Text>
    </View>
  );
}
```

In this example, the sum calculation will only run when the numbers array changes, preventing unnecessary recalculations and improving performance.

When to Use useMemo in React Native

Expensive calculations: Use it when your component has computations that are resource-heavy, like processing large datasets, filtering, sorting, etc.

Re-render optimizations: If certain parts of your component rely on derived values that don’t change often, useMemo can prevent re-renders from triggering unnecessary recalculations.

When to Use useMemo in React Native

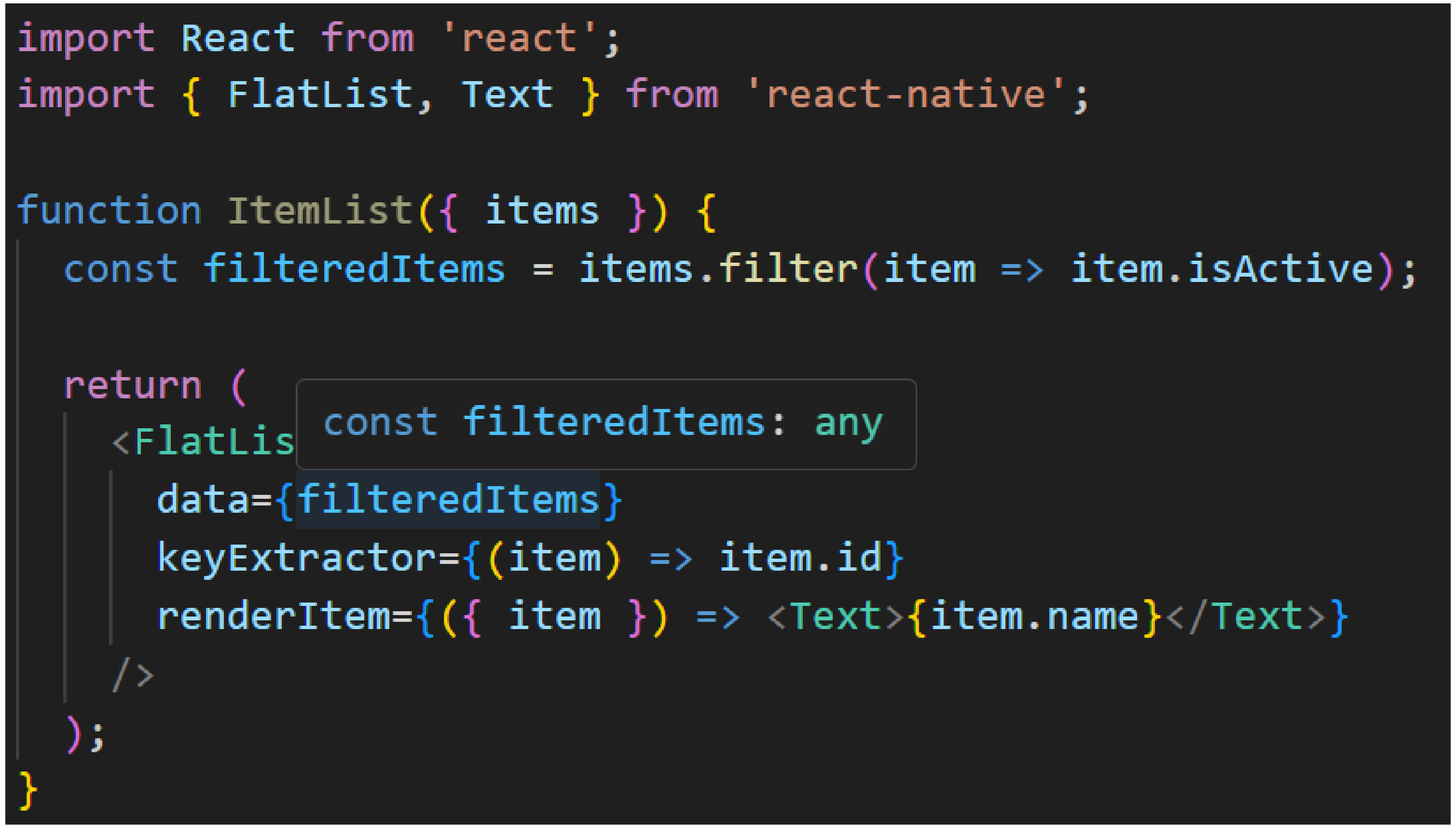
Expensive calculations: Use it when your component has computations that are resource-heavy, like processing large datasets, filtering, sorting, etc.

Re-render optimizations: If certain parts of your component rely on derived values that don’t change often, useMemo can prevent re-renders from triggering unnecessary recalculations.

Example: Optimizing a FlatList-Without useMemo

Imagine you're rendering a large list of items using FlatList, and you need to filter the list before rendering it.

Here, the filtering operation runs on every render, even when the items array hasn't changed.

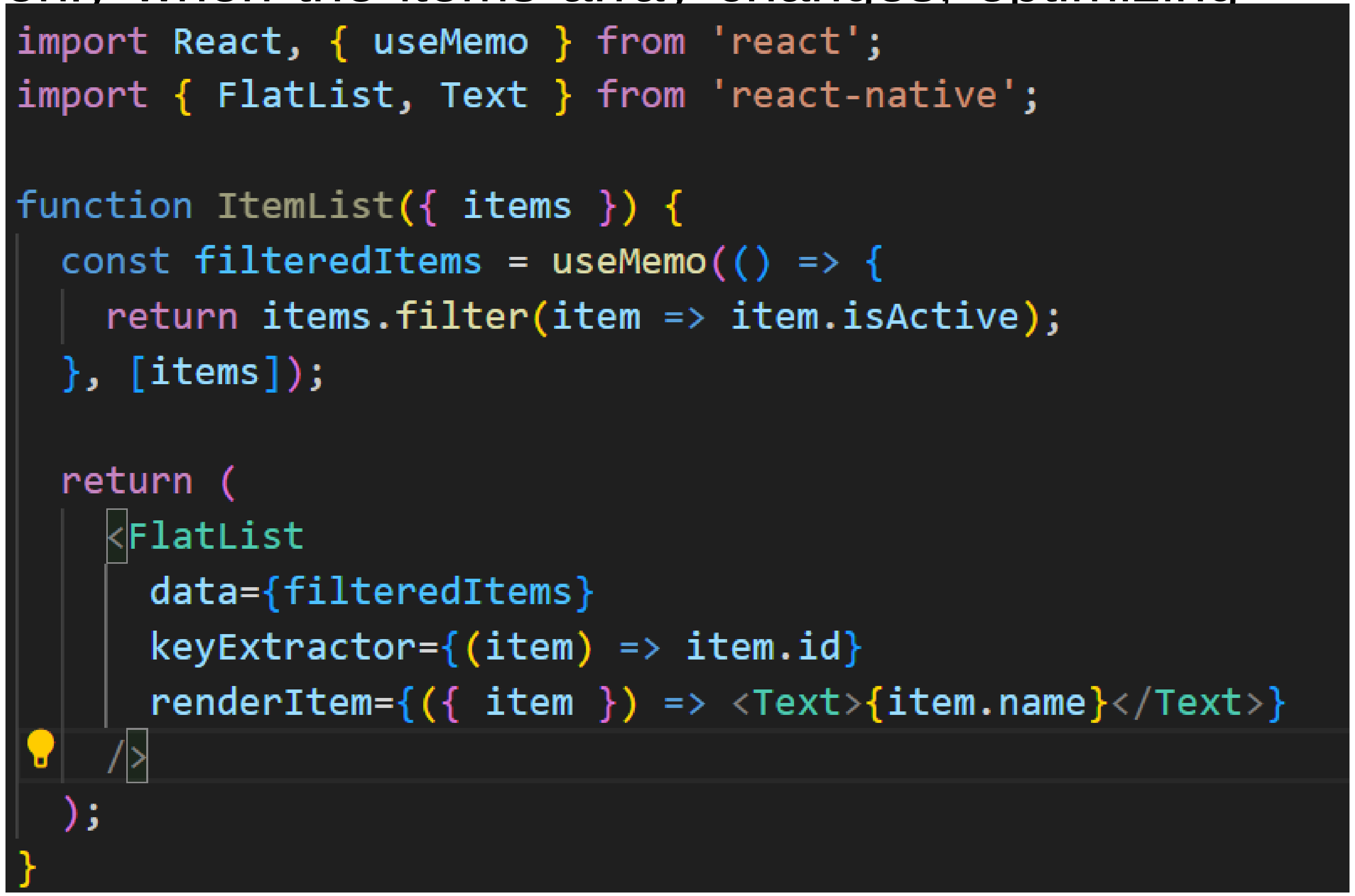


```
import React from 'react';
import { FlatList, Text } from 'react-native';

function ItemList({ items }) {
  const filteredItems = items.filter(item => item.isActive);

  return (
    <FlatLis
      data={filteredItems}
      keyExtractor={(item) => item.id}
      renderItem={({ item }) => <Text>{item.name}</Text>}
    />
  );
}
```

```
const filteredItems: any
```

Example: Optimizing a FlatList-With useMemo

Now, filteredItems is recalculated only when the items array changes, optimizing the list rendering.



```
import React, { useMemo } from 'react';
import { FlatList, Text } from 'react-native';

function ItemList({ items }) {
  const filteredItems = useMemo(() => {
    return items.filter(item => item.isActive);
  }, [items]);

  return (
    <FlatList
      data={filteredItems}
      keyExtractor={(item) => item.id}
      renderItem={({ item }) => <Text>{item.name}</Text>}
    />
  );
}
```

When to Use useMemo in React Native

Expensive calculations: Use it when your component has computations that are resource-heavy, like processing large datasets, filtering, sorting, etc.

Re-render optimizations: If certain parts of your component rely on derived values that don’t change often, useMemo can prevent re-renders from triggering unnecessary recalculations.

When Not to Use useMemo in React Native

Avoid premature optimization: If the calculation is simple and doesn’t have a noticeable performance impact, adding useMemo can make the code more complex without benefits.

Overuse: Overusing **useMemo** across a RN project can add unnecessary complexity. Reserve it for scenarios where performance is noticeably impacted.

In summary, **useMemo** is a useful tool in RN to ensure efficient recalculations, especially in resource-intensive operations. Its purpose and behavior are identical to how it's used in React for web, with the added emphasis on mobile performance.

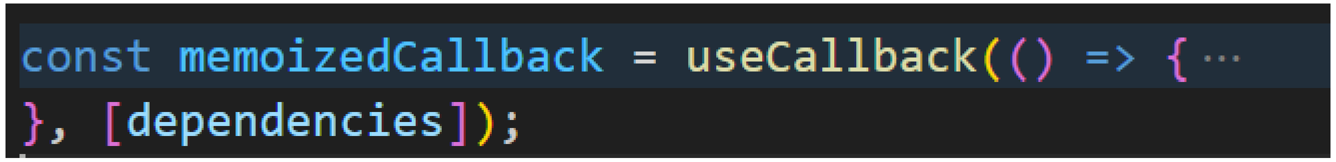
useCallback

In React Native, **useCallback** is a hook similar to useMemo, but instead of memoizing the result of a computation, it memoizes a function. It ensures that the same function reference is used across renders, which can be crucial for preventing unnecessary re-renders of components or for optimizing performance when passing callbacks down to child components.

What is useCallback?

useCallback returns a memoized version of a callback function that only changes if one of its dependencies changes. This can be useful when you need to pass a stable function reference to child components, preventing them from re-rendering unnecessarily.

Syntax


```
const memoizedCallback = useCallback(() => {…
}, [dependencies]);
```

Callback function: The function to be memoized.

[dependencies]: An array of dependencies. The callback will only be updated when one or more of these dependencies change.

Why Use useCallback in React Native?

React Native's rendering system is similar to React's, where unnecessary re-renders of components or unnecessary re-execution of functions can impact performance, particularly in mobile environments where resources are constrained. useCallback helps to:

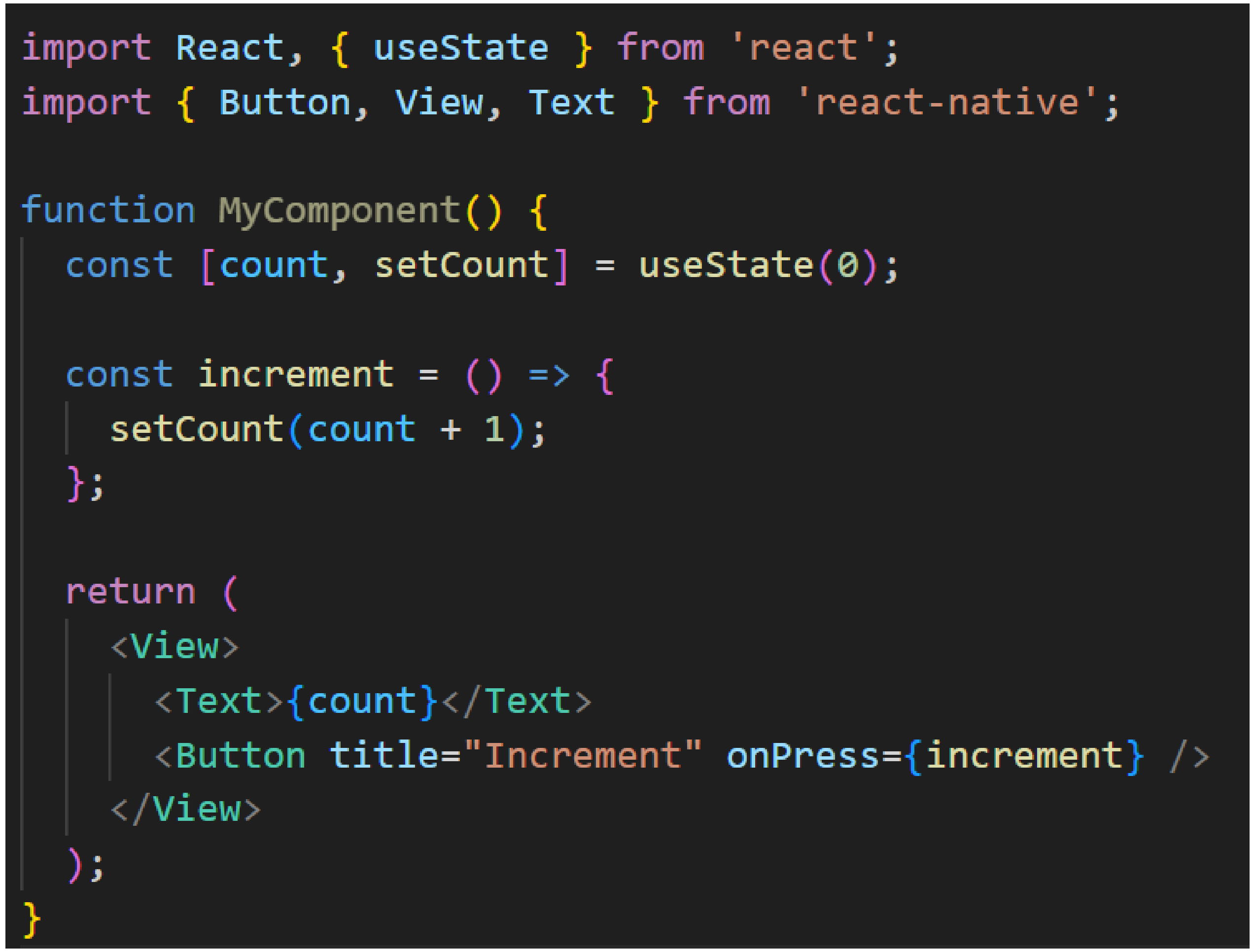
1. Prevent unnecessary re-renders by ensuring that the same function reference is passed to child components or event handlers.
2. Optimize performance by caching the function reference unless the dependencies change.

When to Use useCallback

1. **When passing functions as props to child components:** Child components may re-render unnecessarily if the parent passes a new function reference on every render. **useCallback** ensures the same function reference is used unless dependencies change.
2. **Expensive callback functions:** If the function is computationally expensive or affects rendering performance, **useCallback** helps reduce recalculations.

Example Use Case in React Native-Without useCallback

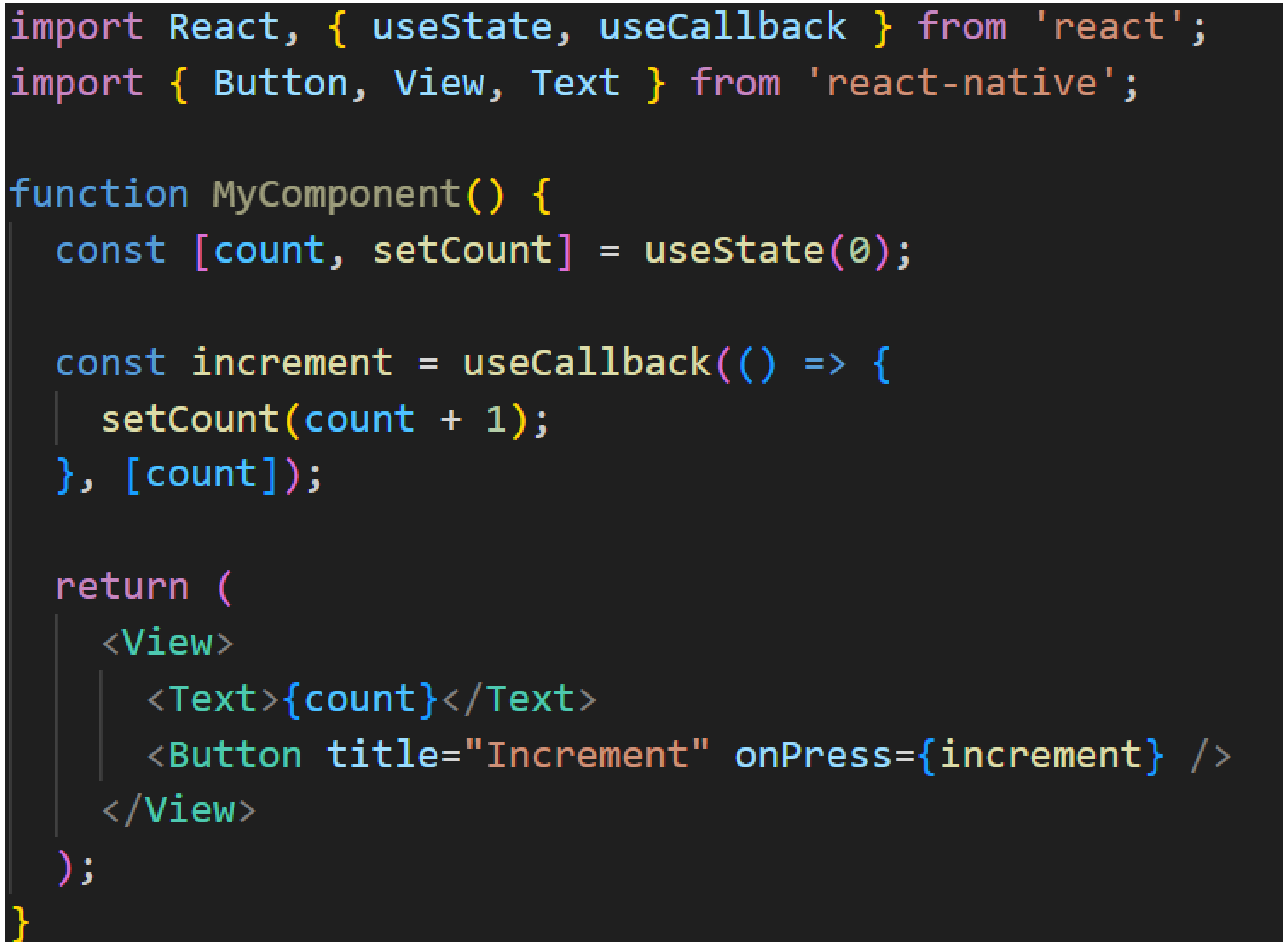
In this case, every time the component re-renders, the **increment function is recreated**, even though the logic hasn't changed.



```jsx
import React, { useState } from 'react';
import { Button, View, Text } from 'react-native';

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

  const increment = () => {
    setCount(count + 1);
  };

  return (
    <View>
      <Text>{count}</Text>
      <Button title="Increment" onPress={increment} />
    </View>
  );
}
```

Example Use Case in React Native-With useCallback

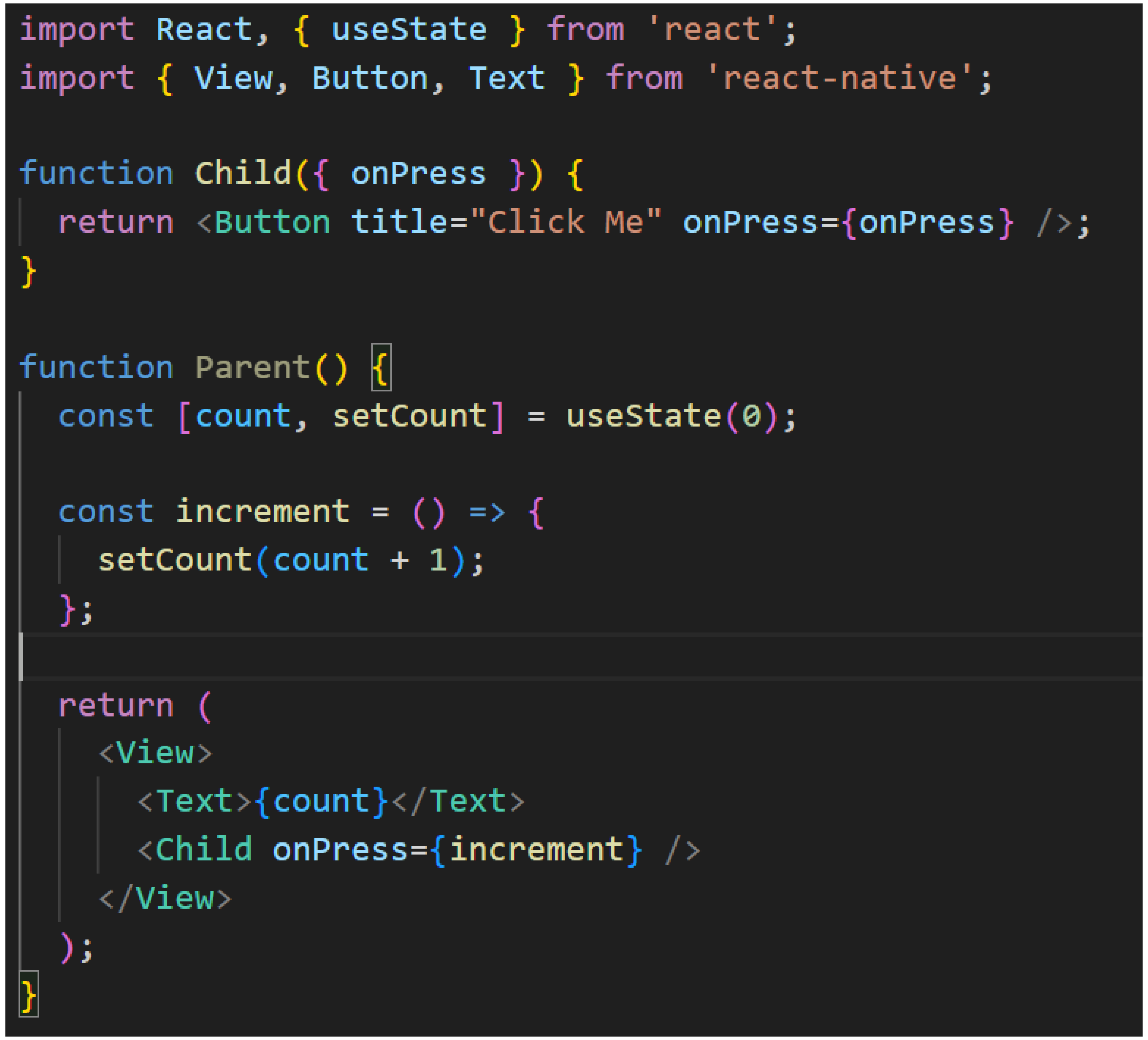
Now, the increment function is memoized and only recreated when the count state changes. This is especially useful when passing the increment function as a prop to child components.



```
import React, { useState, useCallback } from 'react';
import { Button, View, Text } from 'react-native';

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

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

  return (
    <View>
      <Text>{count}</Text>
      <Button title="Increment" onPress={increment} />
    </View>
  );
}
```

Optimizing Child Components–Without useCallback

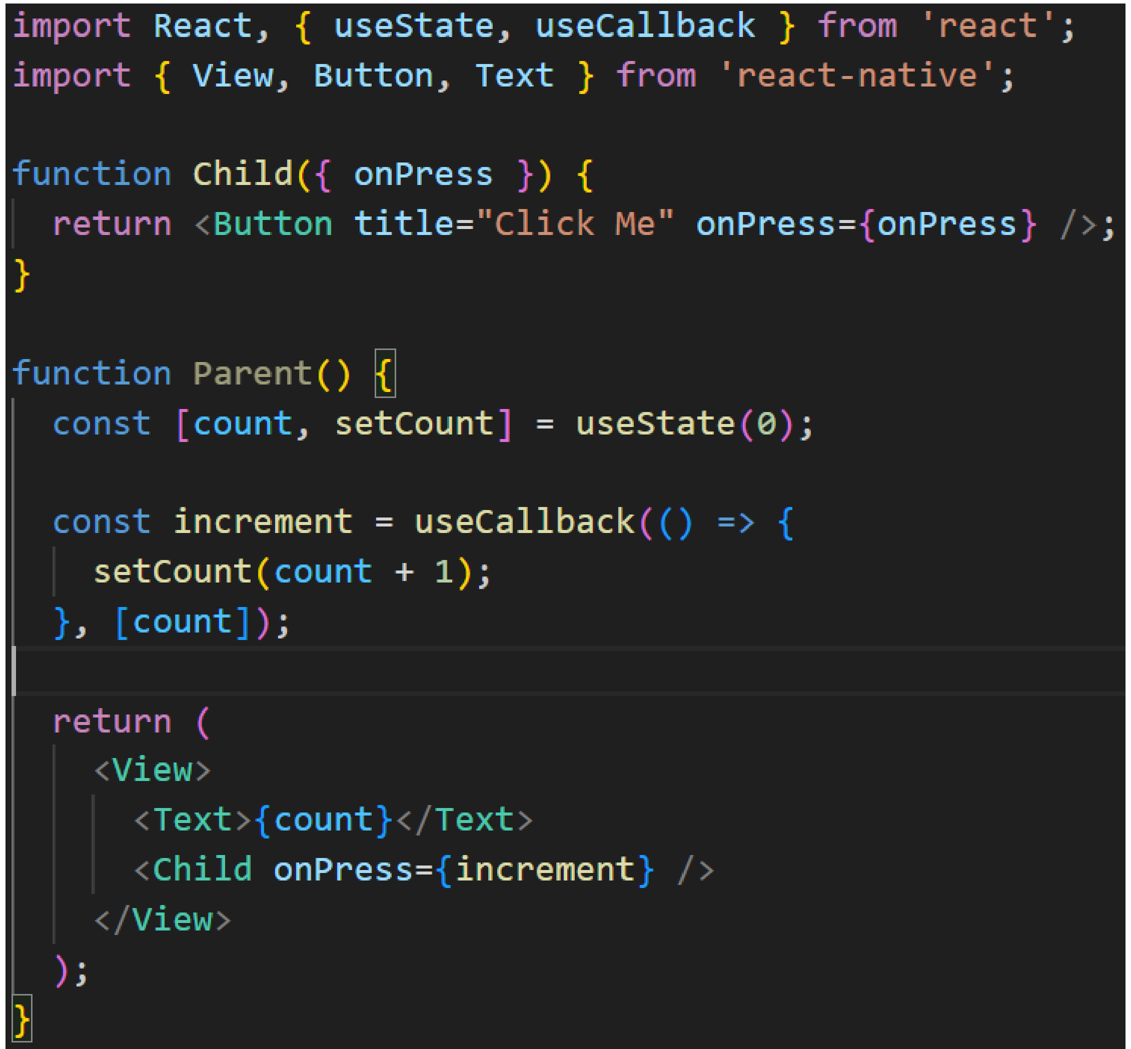
In this example, every time Parent re-renders, the increment function is recreated, causing the Child component to re-render even if it doesn’t need to.



```
import React, { useState } from 'react';
import { View, Button, Text } from 'react-native';

function Child({ onPress }) {
  return <Button title="Click Me" onPress={onPress} />;
}

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

  const increment = () => {
    setCount(count + 1);
  };

  return (
    <View>
      <Text>{count}</Text>
      <Child onPress={increment} />
    </View>
  );
}
```

Example Use Case in React Native-With useCallback

Now, the Child component will only re-render when the increment function reference changes, improving performance by avoiding unnecessary renders



```
import React, { useState, useCallback } from 'react';
import { View, Button, Text } from 'react-native';

function Child({ onPress }) {
  return <Button title="Click Me" onPress={onPress} />;
}

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

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

  return (
    <View>
      <Text>{count}</Text>
      <Child onPress={increment} />
    </View>
  );
}
```

When Not to Use useCallback

1. **Premature optimization:** If the function you're memoizing is not causing performance problems, **useCallback** might add unnecessary complexity.
2. **Simple event handlers:** For basic event handlers (like onClick or onPress), the performance gains might be negligible.

Conclusion

useCallback in React Native is particularly useful for:

1. **Optimizing performance:** When functions are passed as props or used in event handlers, useCallback helps ensure that the same function reference is used across renders unless the dependencies change.
2. **Preventing unnecessary re-renders:** It works well when combined with components that only re-render when the props or state changes, such as React.memo or similar optimization techniques in React Native.

React Custom Hooks

A custom hook in React Native (or React in general) is a reusable function that encapsulates logic and state, allowing you to share it across different components.

Custom hooks allow you to abstract complex logic out of components, making your code cleaner and more maintainable.

What is a Custom Hook?

A custom hook is a **JavaScript function** whose name starts with use (to follow React’s hook convention) and can call other hooks inside it. It allows you to **reuse stateful logic across multiple components** without needing to duplicate code. Custom hooks can use any of the built-in hooks like **useState, useEffect, useCallback**, etc., to implement shared behavior that can then be **reused** in different components.

Syntax of a Custom Hook

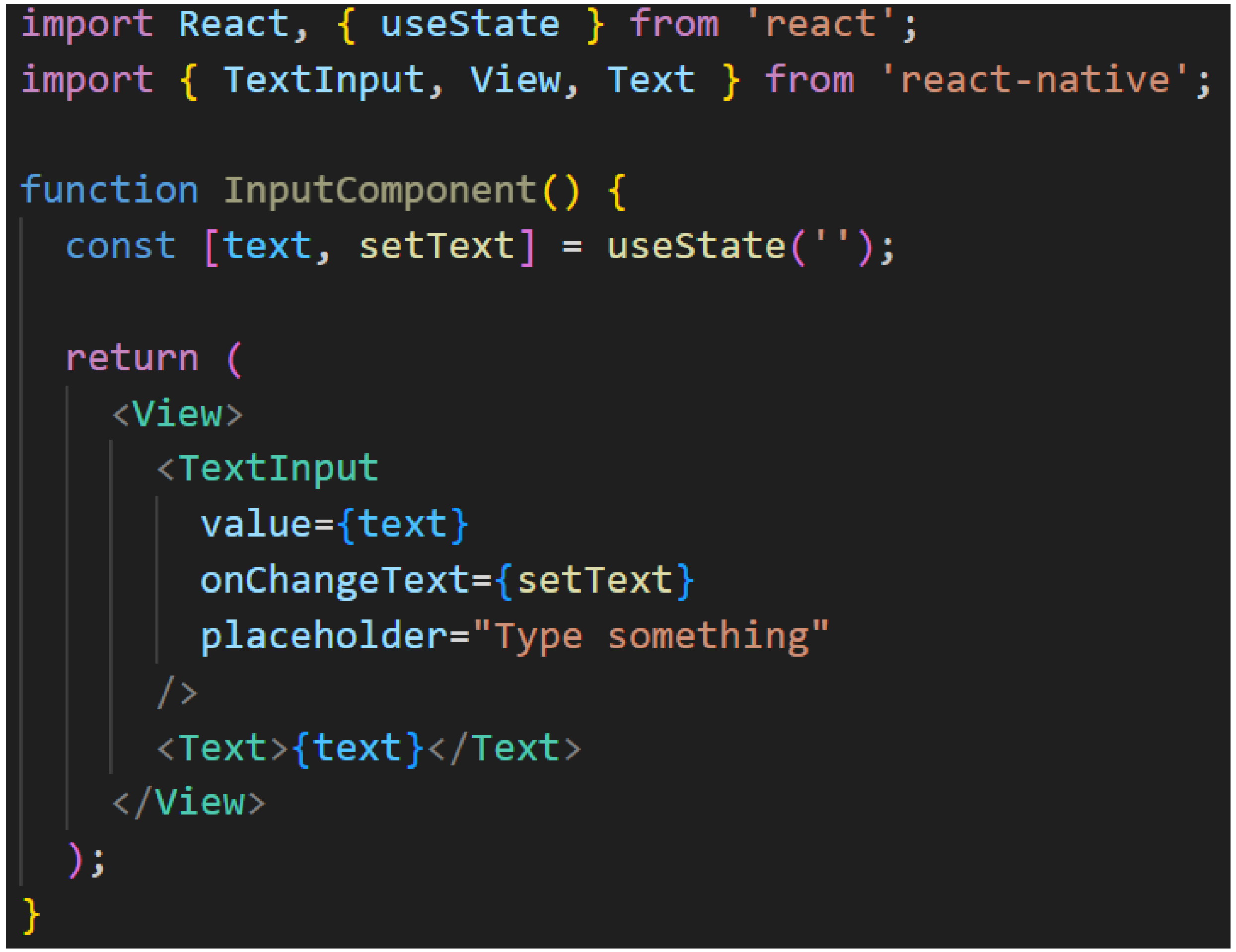
A custom hook is just a regular function that:

- Always starts with use (e.g., **useCustomHook**).
- Can call other React hooks inside it.
- Encapsulates r**eusable logic** that multiple components can use.

Example: Simple Custom Hook for Managing Input

Without a Custom Hook

- If you have multiple input fields, you'll end up duplicating the state management logic in each component. This is where a custom hook comes in handy.

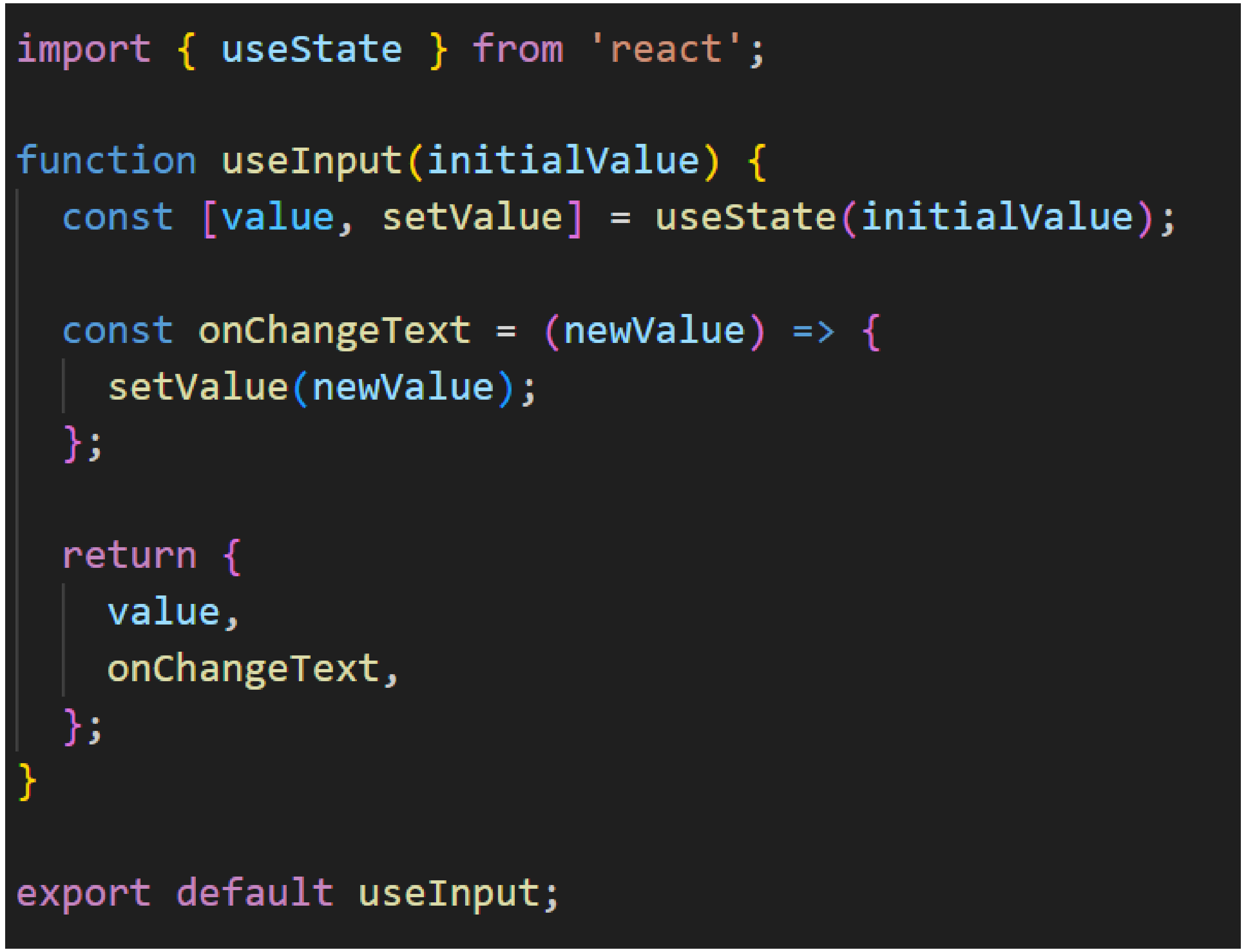


```
import React, { useState } from 'react';
import { TextInput, View, Text } from 'react-native';

function InputComponent() {
  const [text, setText] = useState('');

  return (
    <View>
      <TextInput
        value={text}
        onChangeText={setText}
        placeholder="Type something"
      />
      <Text>{text}</Text>
    </View>
  );
}
```

Example: Simple Custom Hook for Managing Input

With a Custom Hook

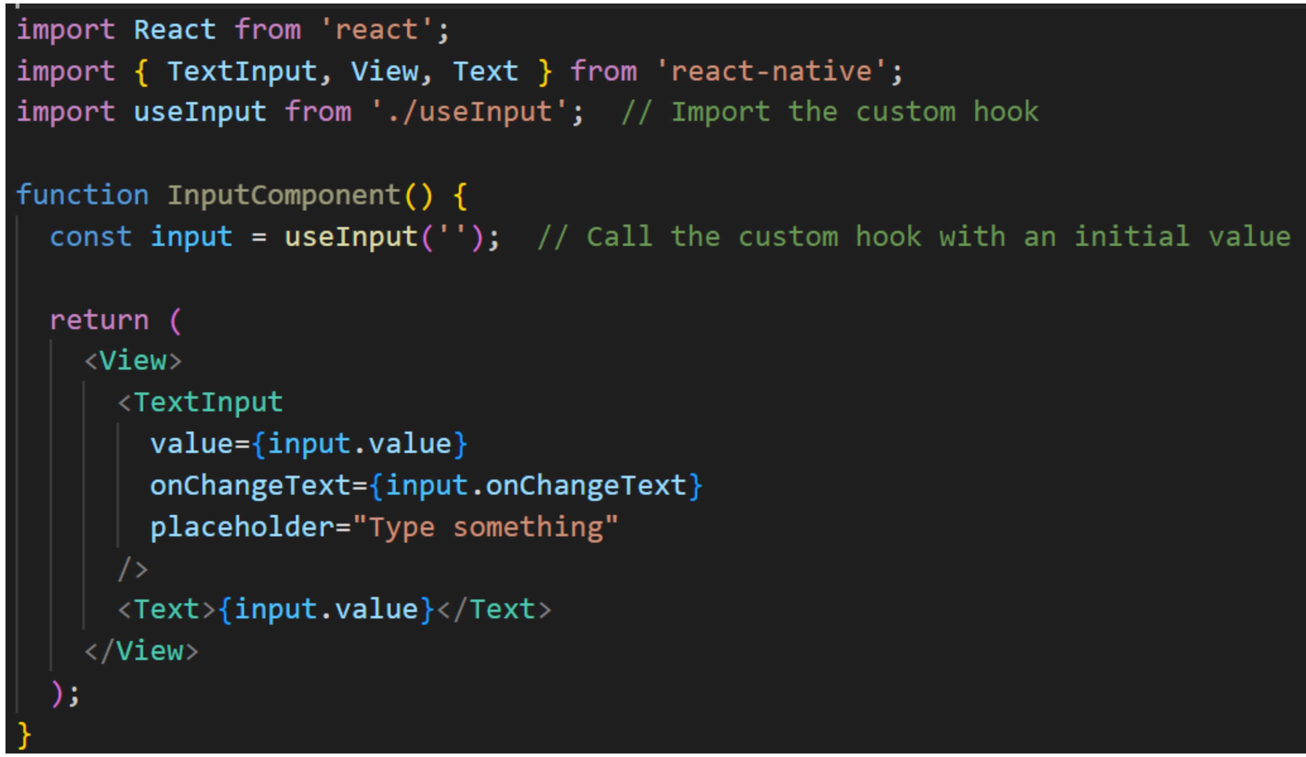
- You can create a custom hook called **useInput** to encapsulate the state and the logic for handling text inputs.
- **Now, you can reuse this custom hook in multiple components.**



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

function useInput(initialValue) {
  const [value, setValue] = useState(initialValue);

  const onChangeText = (newValue) => {
    setValue(newValue);
  };

  return {
    value,
    onChangeText,
  };
}

export default useInput;
```

Using the Custom Hook in Components

Now, the **useInput** hook encapsulates all the input logic, allowing you to reuse it across multiple components. Each instance of the useInput hook is independent, so you can manage the state for different inputs without duplicating code.



```
import React from 'react';
import { TextInput, View, Text } from 'react-native';
import useInput from './useInput';  // Import the custom hook

function InputComponent() {
  const input = useInput('');  // Call the custom hook with an initial value

  return (
    <View>
      <TextInput
        value={input.value}
        onChangeText={input.onChangeText}
        placeholder="Type something"
      />
      <Text>{input.value}</Text>
    </View>
  );
}
```

Benefits of Custom Hooks

Reusability: You can extract commonly used logic (like fetching data, managing form inputs, or managing local state) into a custom hook, which can be reused in multiple components.

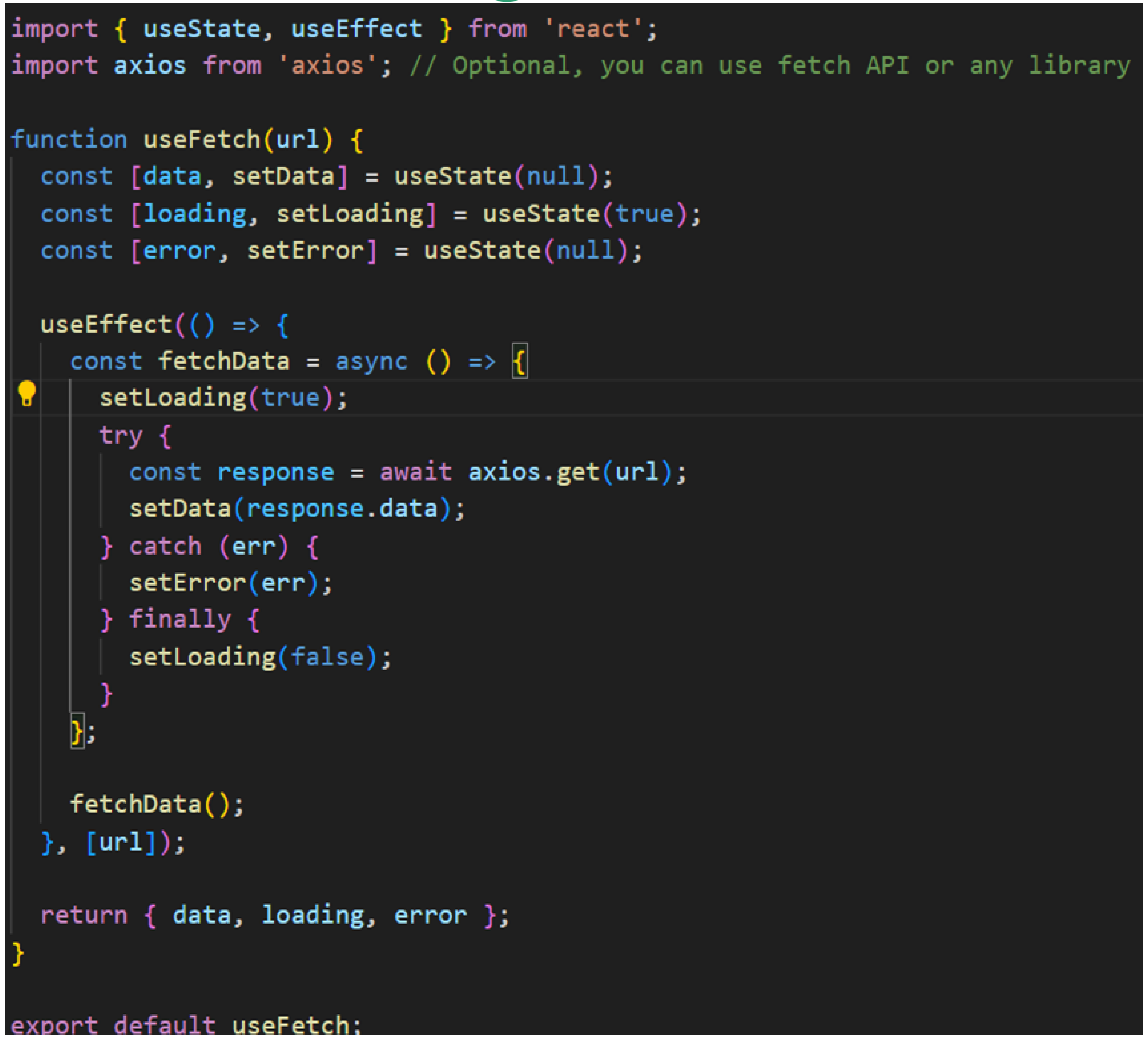
Cleaner Components: By moving logic out of the components and into custom hooks, components can focus more on UI, making them easier to read and maintain.

Encapsulation: Custom hooks encapsulate logic, keeping your components clean and reducing code duplication.

Composability: Hooks are composable, meaning you can create more complex custom hooks by combining multiple custom or built-in hooks.

Example: Custom Hook for Fetching Data

Let’s create a custom hook that fetches data from an API and returns the loading state, error, and the data itself. This can be reused in any component that needs to fetch data.



```javascript
import { useState, useEffect } from 'react';
import axios from 'axios'; // Optional, you can use fetch API or any library

function useFetch(url) {
  const [data, setData] = useState(null);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    const fetchData = async () => {
      setLoading(true);
      try {
        const response = await axios.get(url);
        setData(response.data);
      } catch (err) {
        setError(err);
      } finally {
        setLoading(false);
      }
    };

    fetchData();
  }, [url]);

  return { data, loading, error };
}

export default useFetch;
```

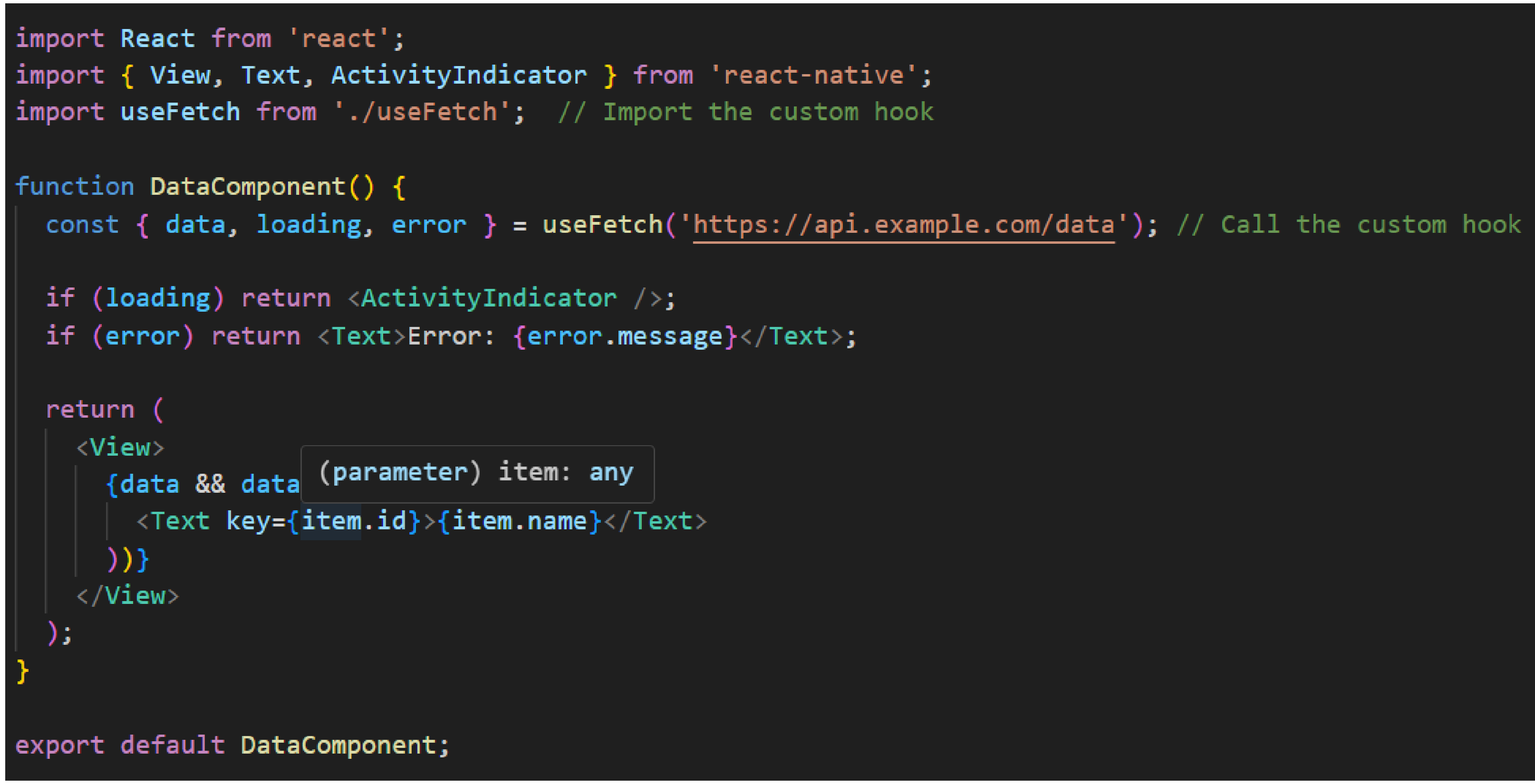
Using the Custom Hook in a Component

Explanation:

useFetch custom hook encapsulates the logic for fetching data from an API.

It handles loading, error states, and stores the fetched data.

The custom hook is reusable and can be used with any API endpoint.



```
import React from 'react';
import { View, Text, ActivityIndicator } from 'react-native';
import useFetch from './useFetch';  // Import the custom hook

function DataComponent() {
  const { data, loading, error } = useFetch('https://api.example.com/data'); // Call the custom hook

  if (loading) return <ActivityIndicator />;
  if (error) return <Text>Error: {error.message}</Text>;

  return (
    <View>
      {data && data
        <Text key={item.id}>{item.name}</Text>
      ))}
    </View>
  );
}

export default DataComponent;
```

Guidelines for Creating Custom Hooks

Start with use: Always name custom hooks starting with use to follow React's convention and ensure that React can understand and enforce hook rules (like only calling hooks inside function components or other hooks).

Encapsulate Reusable Logic: Move logic that is used across multiple components (like state management, API calls, or side effects) into a custom hook to avoid duplication.

Combine Other Hooks: Custom hooks can use other hooks like useState, useEffect, useContext, etc. This allows you to compose and reuse logic effectively.

Return Values: A custom hook can return any value (state, functions, etc.), which can then be destructured or accessed in the components where it's used.

THANK YOU!