React Hooks

Kavya Srinivasa

React Hooks

- React Hooks are functions that provide a way to use state and other React features in functional components.
- Before hooks, stateful logic was limited to class components, which could lead to complex and nested code structures.
- With hooks, functional components can now manage state and lifecycle methods just like class components.

Hooks Rules

- Hooks can only be called inside React function components.
- Hooks can only be called at the top level of a component.
- Hooks cannot be conditional

useState Hook

- The ‘useState’ hook is the most basic and commonly used hook in React.
- It allows you to add state to your functional components.
- The hook returns a stateful value and a function to update that value.

Syntax

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

State: This is the current state value that we want to track.

setState: This is a function used to update the state.

initialState :The initial value set to the state

Lets code

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

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

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

  return (
    <div>
      <h1>Count: {count}</h1>
      <button
onClick={increment}>Increment</button>
    </div>
  );
};
export default Counter;
```

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

const useStateExample= () => {
  const [inputValue, setInputValue] = useState(“Galaxe solutions");

  let handleChange= (event) => {
    const newValue = event.target.value;
    setInputValue(newValue);
  };

  return (
    <div>
      <input placeholder="enter something..." onChange={handleChange} />
      {inputValue}
    </div>
  );
};

export default useStateExample;
```

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

const FormData = () => {
  const [formData, setFormData] = useState({
    username: '',
    email: '',
    password: '',
  });

  const handleChange = (e) => {
    setFormData({
      ...formData,
      [e.target.name]: e.target.value,
    });
  };

  return (
    <form>
      <input type="text" name="username" value={formData.username} onChange={handleChange} />
      <input type="email" name="email" value={formData.email} onChange={handleChange} />
      <input type="password" name="password" value={formData.password} onChange={handleChange} />
    </form>
  );
};
export default FormData;
```

Rules of Using useState

- The useState hook must be called at the top level of the functional component.
- The order of hooks must be the same in each render call.

Benefits of useState

- Simplifies state management in functional components.
- Reduces the amount of code compared to using class components.
- Improves code readability and maintainability.
- No need to worry about the "this" keyword, as in class components.

useReducer Hook

- The 'useReducer' hook provides an alternative way to handle more complex state and logic in functional components
- The useReducer hook is a powerful tool in React that allows us to manage state in a more organized and structured way.
- It is an alternative to useState and is particularly useful when the state has complex transitions that involve multiple sub-values.
- useReducer follows the same principles as the Redux library, where state transitions are determined by a function called a "reducer."

Syntax

```
const [state, dispatch] = useReducer(reducer, initialState);
```

State: The current state value, similar to the state returned by useState

setState: A function that allows you to dispatch actions to trigger state transitions.

initialState :The initial value set to the state

Reducer : It takes two arguments: the current state and an action object that describes the state change. The reducer's responsibility is to return the new state based on the action type.

```
const reducer = (state, action) => {
  switch (action.type) {
    case ‘type1':
      return { corresponding action to type 1};
    case ‘type2':
      return {corresponding action to type 1};
    default:
      return default action;
  }
};
```

Lets code

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

const reducer = (state, action) => {
  switch (action.type) {
    case "INCREMENT":
      return { count: state.count + 1, showText: state.showText };
    case "toggleShowText":
      return { count: state.count, showText: !state.showText };
    default:
      return state;
  }
};

const ReducerTutorial = () => {
  const [state, dispatch] = useReducer(reducer, { count: 0, showText: true });
```

```
return (
    <div>
      <h1>{state.count}</h1>
      <button
        onClick={() => {
          dispatch({ type: "INCREMENT" });
          dispatch({ type: "toggleShowText" });
        }}
      >
        Click Here
      </button>

      {state.showText && <p>This is a text</p>}
    </div>
  );
};

export default ReducerTutorial;
```

Benefits of useReducer

- Helps manage complex state transitions and logic more effectively.
- Provides a predictable way to update state based on actions.

When to use UseReducer over usestate

- If the state logic is simple, stick to: useState.
- If the state transitions are complex and involve multiple sub-values, consider, useReducer.
- If you find yourself writing multiple useState calls to handle related state, it might be a good candidate for useReducer.

useEffect Hook

- The useEffect hook is used to handle side effects in functional components.
- Side effects include data fetching, subscriptions, or manually changing the DOM.

Syntax

```
useEffect(() => {

}, [dependency1, dependency2]);
```

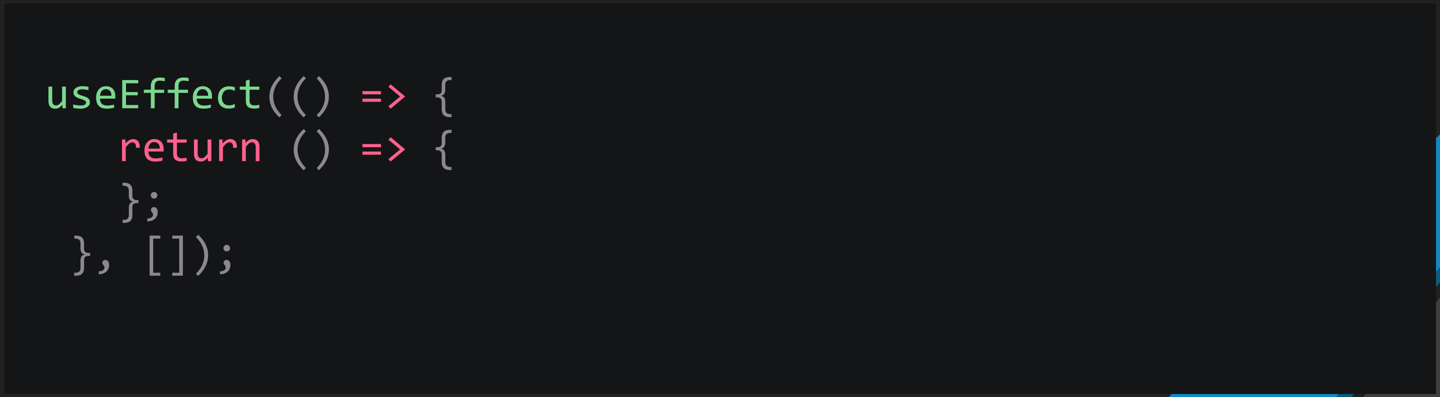
- The **useEffect** hook takes two arguments: a **callback** function and an optional **array of dependencies**.
- The callback function represents the side effect you want to perform.
- The dependencies array is used to control when the side effect runs. If any value in the array changes, the side effect is triggered.

Typical Use Cases

- **Data Fetching**: Fetching data from APIs and updating the state with the fetched data.
- **Subscriptions**: Setting up and cleaning up subscriptions to events or services.
- **DOM Manipulation**: Changing the DOM manually when necessary.
- **Timers and Intervals**: Managing timers and intervals in your component.

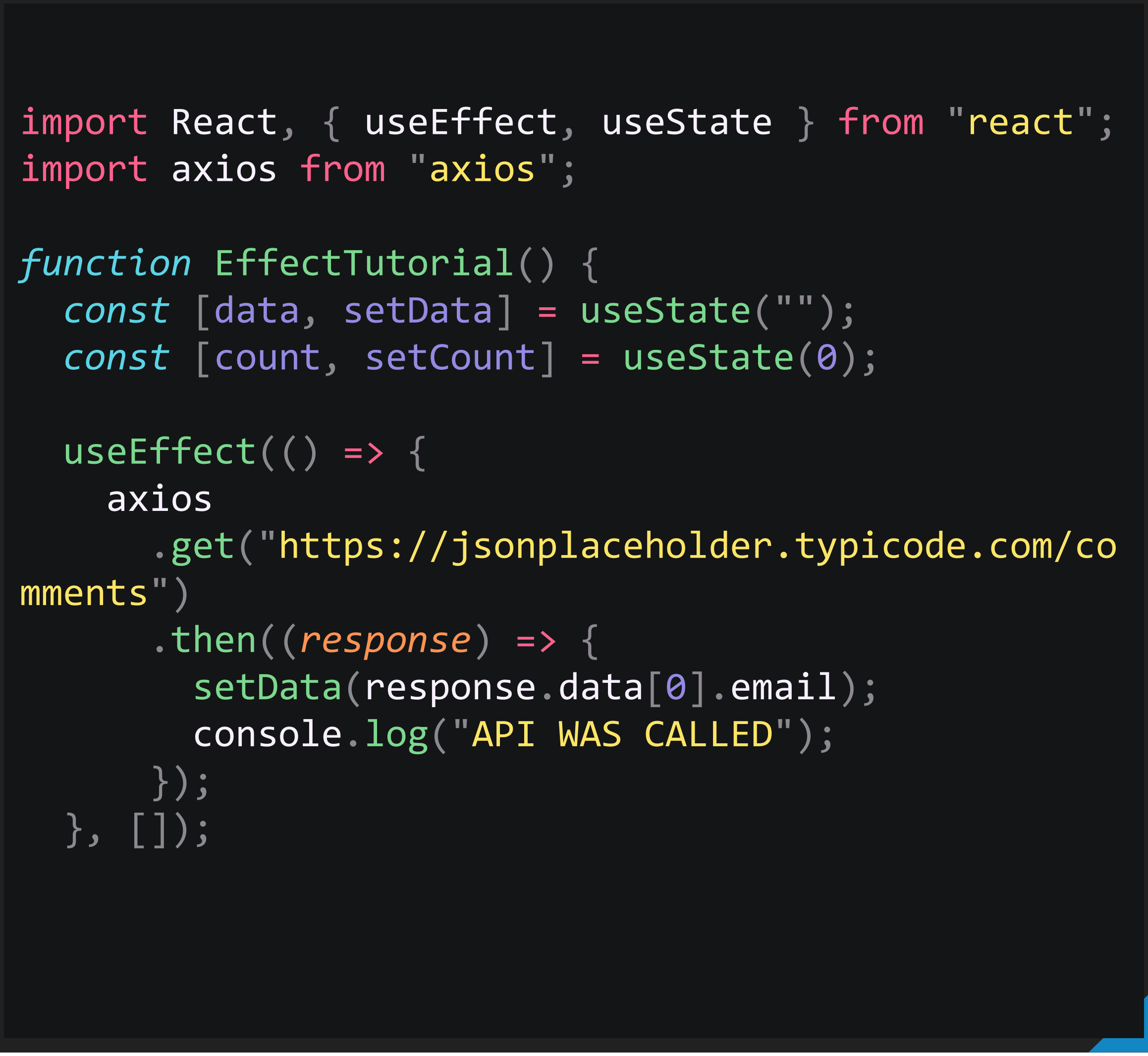
The Cleanup Function

- The callback function returned from useEffect can be used to perform cleanup operations.
- This is especially useful for unsubscribing from event listeners or clearing up resources to avoid memory leaks.

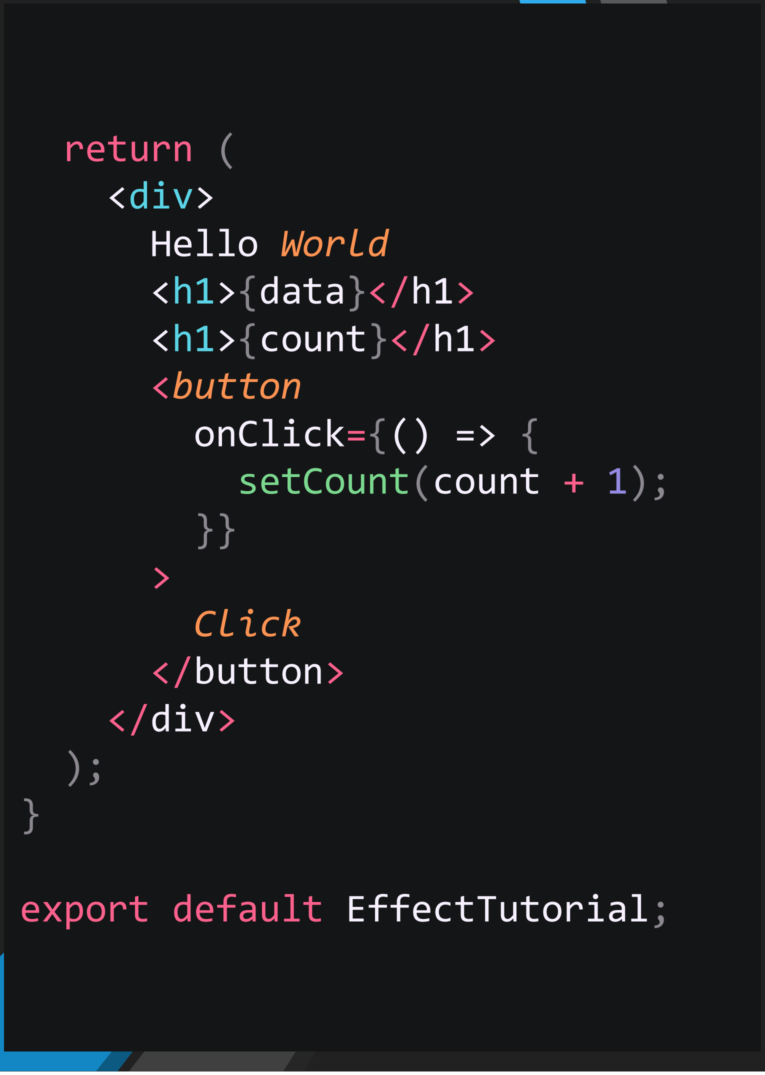


```
useEffect(() => {
   return () => {
   };
 }, []);
```

Lets code



```
import React, { useEffect, useState } from "react";
import axios from "axios";

function EffectTutorial() {
  const [data, setData] = useState("");
  const [count, setCount] = useState(0);

  useEffect(() => {
    axios
      .get("https://jsonplaceholder.typicode.com/co
mments")
      .then((response) => {
        setData(response.data[0].email);
        console.log("API WAS CALLED");
      });
  }, []);
```

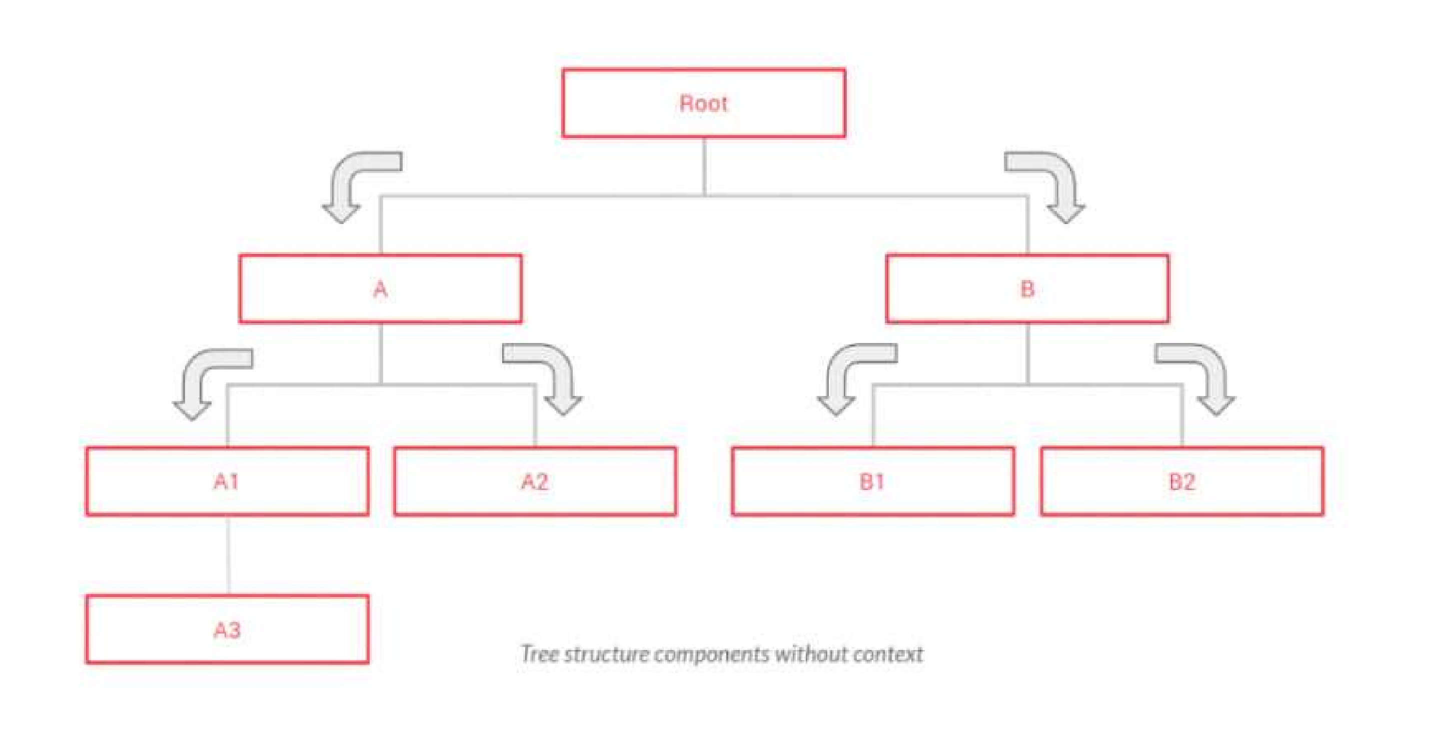


```
  return (
    <div>
      Hello World
      <h1>{data}</h1>
      <h1>{count}</h1>
      <button
        onClick={() => {
          setCount(count + 1);
        }}
      >
        Click
      </button>
    </div>
  );
}

export default EffectTutorial;
```

useContext Hook

- The ‘useContext’ hook is used to consume data from a React context.
- Context provides a way to share data across the component tree without manually passing props down through each level.
- React Context is a way to manage state globally.
- State should be held by the highest parent component in the stack that requires access to the state

Props Drilling



Tree structure components without context

Setting Up a Context

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

const MyContext = createContext();
export default MyContext;
```

Providing Data with the Context Provider

```
import React from 'react';
import MyContext from './MyContext';

const App = () => {
  const sharedData = { username: ‘galaxe@123', theme: 'dark' };

  return (
    <MyContext.Provider value={sharedData}>
      {/* Your component tree goes here */}
    </MyContext.Provider>
  );
};

export default App;
```

Consuming Data with useContext

```
import React, { useContext } from 'react';
import MyContext from './MyContext';

const MyComponent = () => {
  const sharedData = useContext(MyContext);

  return (
    <div>
      <h1>Welcome, {sharedData.username}!</h1>
      <p>Current theme: {sharedData.theme}</p>
    </div>
  );
};
```

Benefits of useContext

- Avoids prop drilling and keeps your code cleaner.
- Simplifies state management for shared data.
- Makes it easy to access global state within any component.

When to use UseContext

- When you have data that needs to be shared across multiple components without passing it explicitly through props
- When you want to avoid the complexity of prop drilling

useRef Hook

- useRef is a built-in React Hook that creates a mutable object called a "ref.“
- Unlike state or props, refs persist across renders and do not trigger re-renders when updated.

Syntax

```
const myRef = useRef(initialValue);
```

Use Cases of useRef

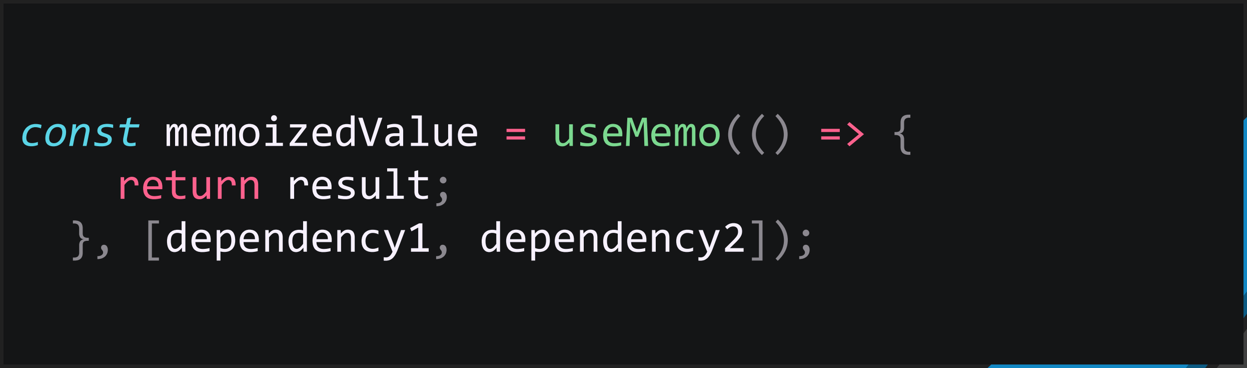
- Referencing DOM Elements: Access and manipulate DOM elements imperatively without triggering re-renders.
- Managing Previous Values: Keep track of previous values of props or state without using state variables.
- Caching Expensive Computations: Cache the results of expensive computations to avoid recomputing on each render.

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

const TextInput = () => {
  const inputRef = useRef();

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

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

useMemo Hook

- useMemo is a built-in React Hook used to memorize the result of expensive function calls.
- It returns the memorized result when the input dependencies remain the same, preventing unnecessary re-computations.

Syntax



```
const memoizedValue = useMemo(() => {
    return result;
  }, [dependency1, dependency2]);
```

Use Cases of useMemo

- Optimizing Expensive Computations: Reduce the computation time for heavy operations like sorting or filtering large datasets.
- Preventing Unnecessary Re-renders: Avoid re-running computations when the input data remains the same between renders.
- Caching Calculations: Cache the result of complex calculations to enhance performance.

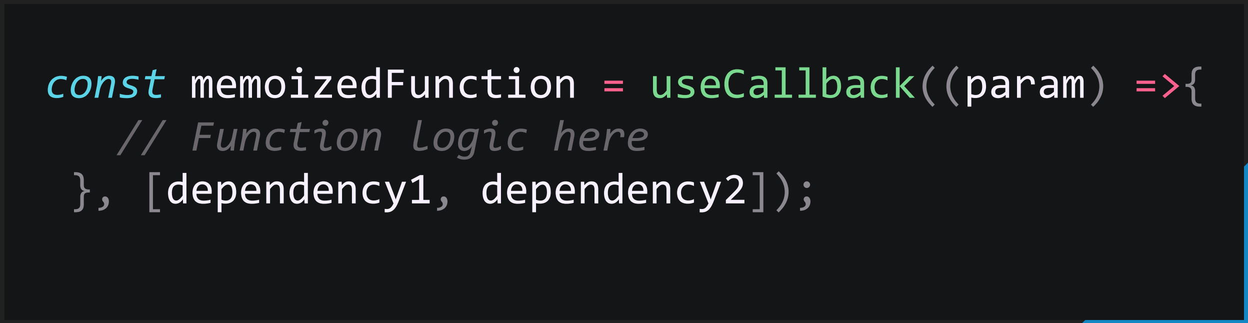
Optimizing Expensive Computations

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

const ExpensiveComputationComponent = ({
data }) => {
  const sortedData = useMemo(() => {
    // Expensive sorting operation
    return data.sort((a, b) => a - b);
  }, [data]);

  return (
    <div>
      {/* Use sortedData */}
    </div>
  );
};
```

useCallback Hook

- useCallback is a built-in React Hook used to memoize functions, preventing unnecessary re-creation of functions on each render.
- It returns a memoized version of the function that only changes if the input dependencies change.

Syntax

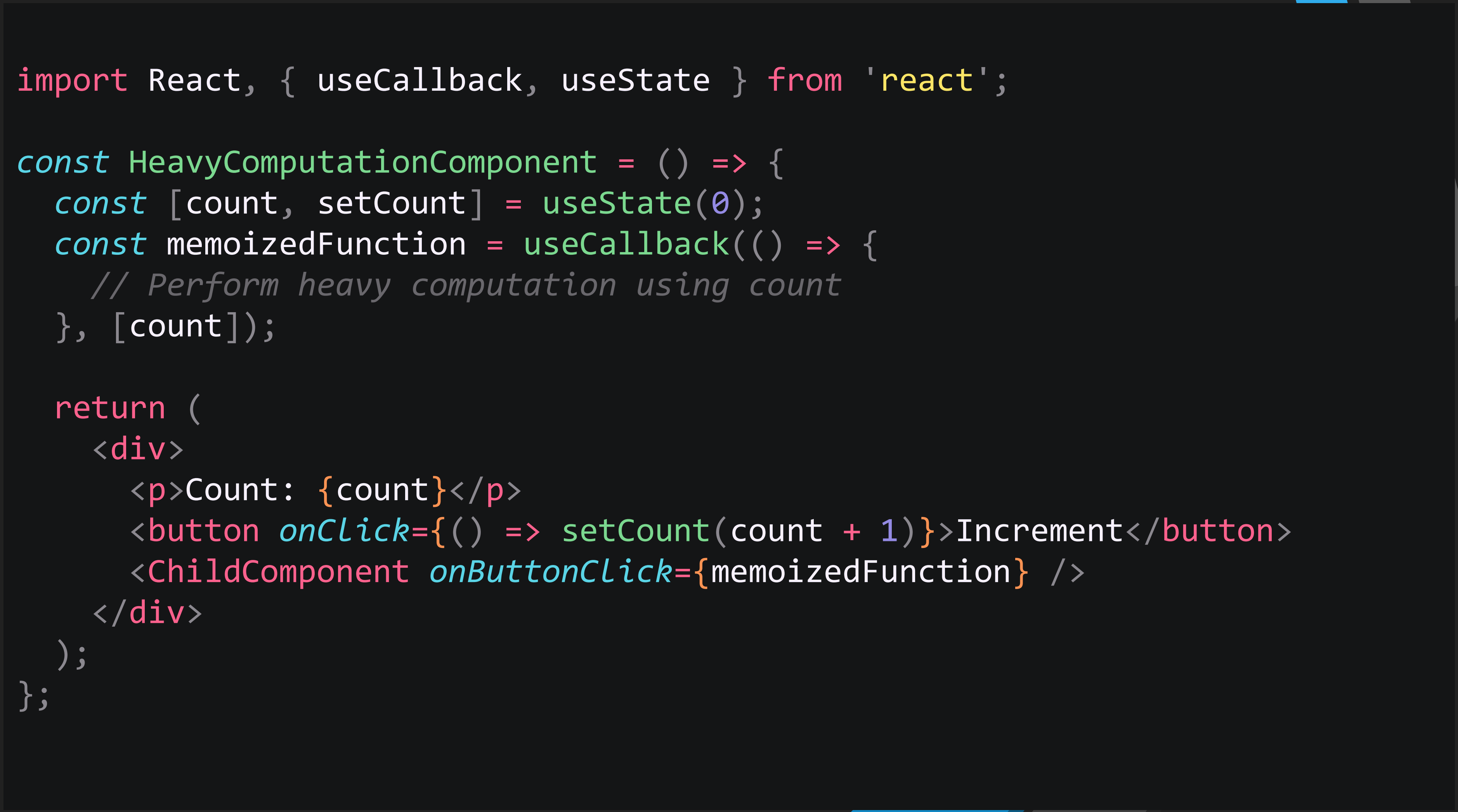


```
const memoizedFunction = useCallback((param) =>{
   // Function logic here
 }, [dependency1, dependency2]);
```

Use Cases of useCallback

- Optimizing Performance: Prevent re-creation of functions, which can lead to improved performance.

- Avoiding Unnecessary Re-renders: Prevent child components from re-rendering if the function reference hasn't changed.

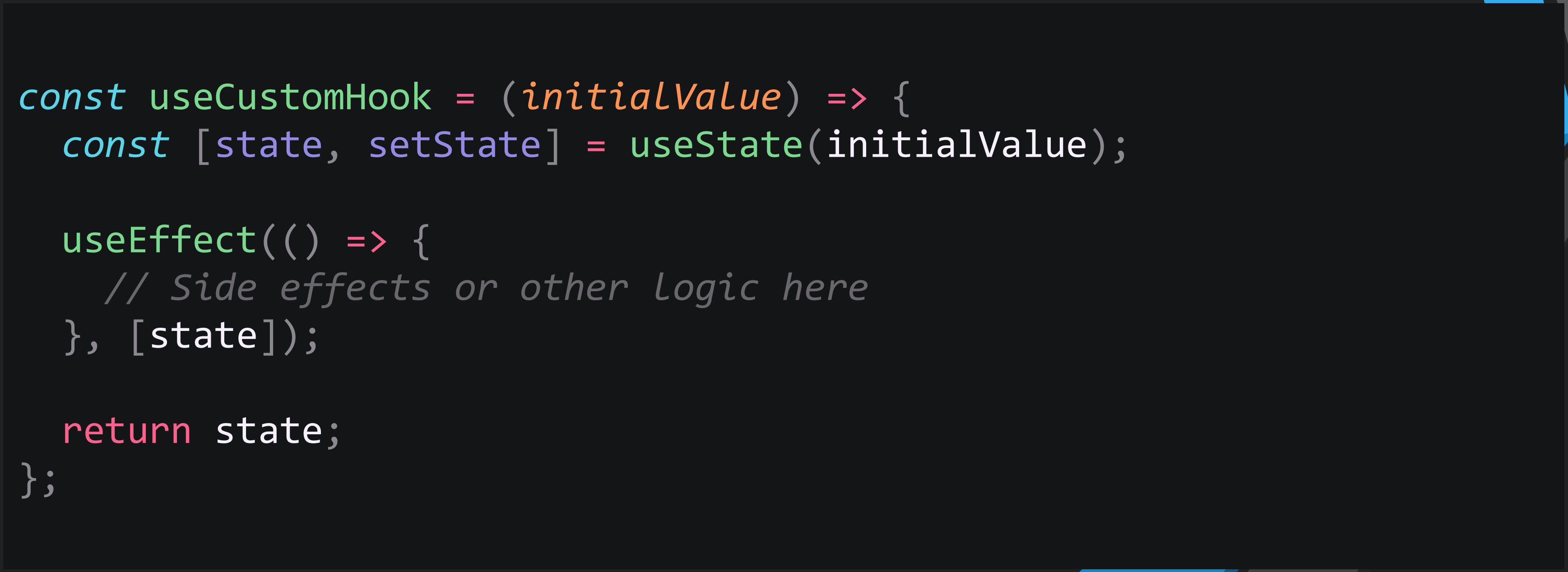


```
import React, { useCallback, useState } from 'react';

const HeavyComputationComponent = () => {
  const [count, setCount] = useState(0);
  const memoizedFunction = useCallback(() => {
    // Perform heavy computation using count
  }, [count]);

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

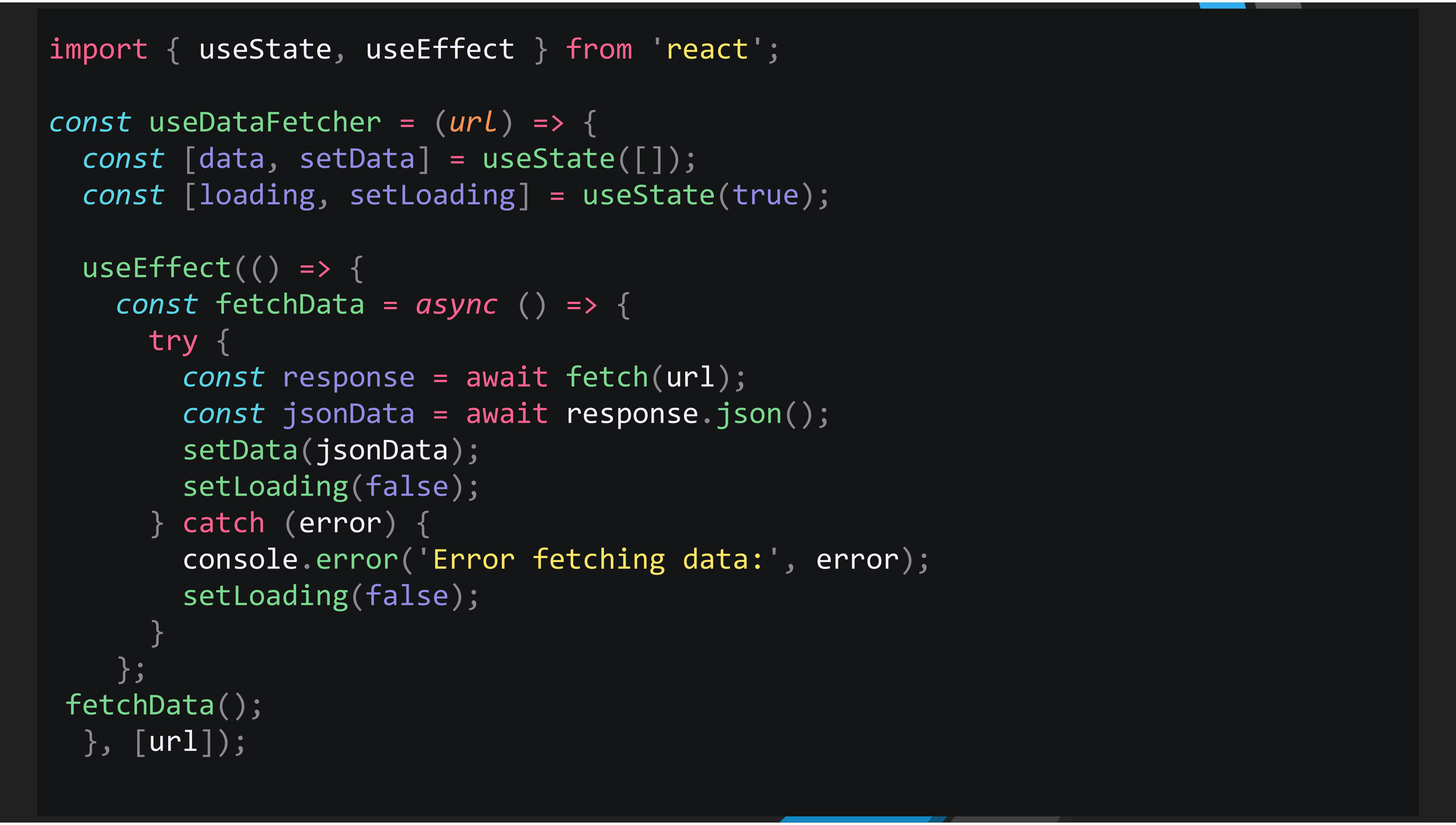
Custom Hook

- Custom Hooks are user-defined hooks that encapsulate and reuse common logic across multiple components.
- They allow us to abstract complex logic into reusable functions, promoting code organization and readability.

syntax



```
const useCustomHook = (initialValue) => {
  const [state, setState] = useState(initialValue);

  useEffect(() => {
    // Side effects or other logic here
  }, [state]);

  return state;
};
```



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

const useDataFetcher = (url) => {
  const [data, setData] = useState([]);
  const [loading, setLoading] = useState(true);

  useEffect(() => {
    const fetchData = async () => {
      try {
        const response = await fetch(url);
        const jsonData = await response.json();
        setData(jsonData);
        setLoading(false);
      } catch (error) {
        console.error('Error fetching data:', error);
        setLoading(false);
      }
    };
 fetchData();
  }, [url]);
```

Thank you