Purwadhika
Digital Technology School

Fullstack Web Development

React Hooks

Job Connector Program

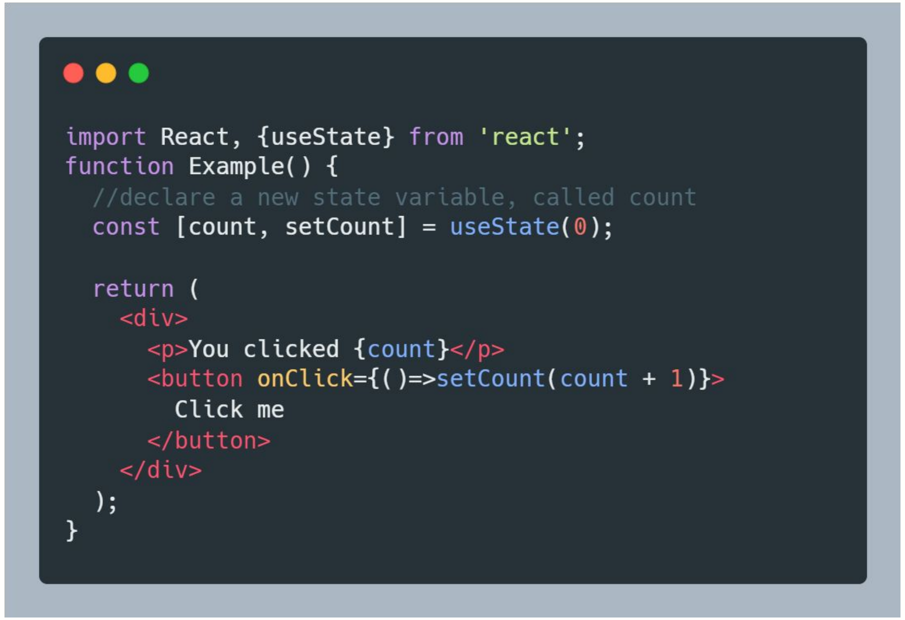
Outline

- Hooks concept
- Common used hooks (useState, useEffect, etc)
- Rules of hooks
- Custom hooks

Hooks Concept

Why we need Hooks:

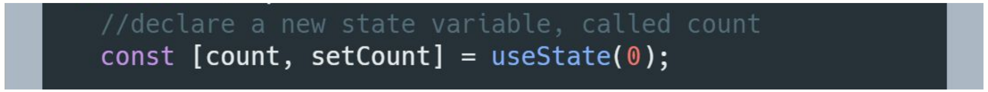
- Hooks allow you to reuse stateful logic without changing your component hierarchy
- Hooks let you split one component into smaller functions based on what pieces are related (such as setting up a subscription or fetching data)
- Hooks let you use more of React's features without classes



```
import React, {useState} from 'react';
function Example() {
  //declare a new state variable, called count
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>You clicked {count}</p>
      <button onClick={()=>setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

useState

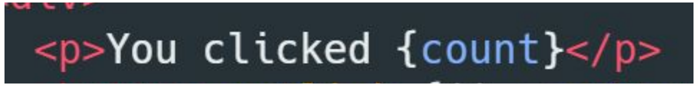
Starts by importing the useState Hook from React, then when we want to declare a new state variable, make a syntax like example below. Count is a state variable and its value is 0 because we define that in parameter useState(0). setCount is a function for update the count state.



```
//declare a new state variable, called count
const [count, setCount] = useState(0);
```

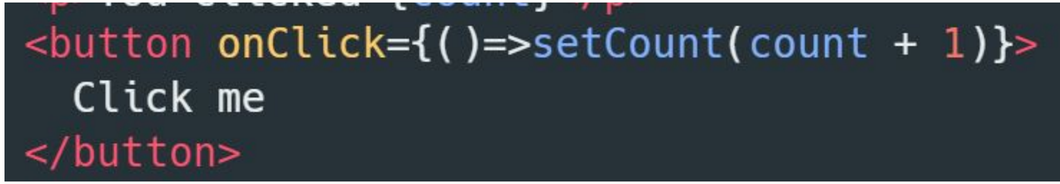
useState

When we want to display the current value of count, we can use count directly:



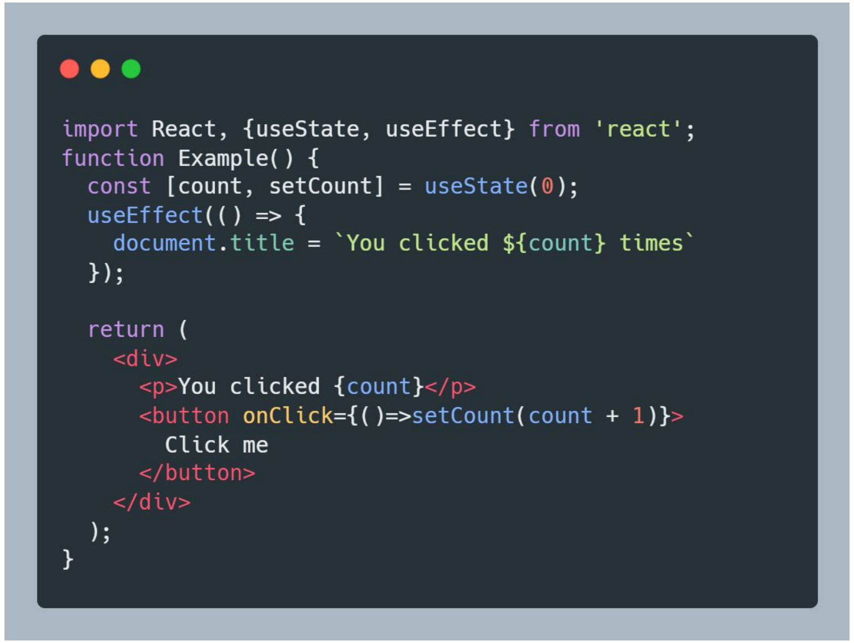
```
<p>You clicked {count}</p>
```

Just call setCount that we defined before, then put new value in parameter's setCount that we want to replace the current value of count.



```
<button onClick={()=>setCount(count + 1)}>
  Click me
</button>
```

useEffect



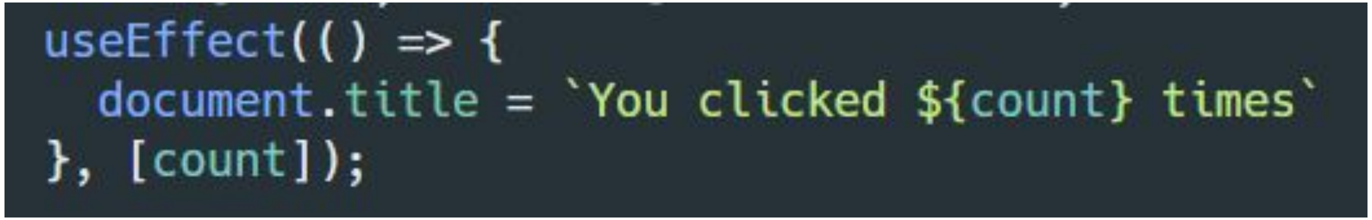
```
import React, {useState, useEffect} from 'react';
function Example() {
  const [count, setCount] = useState(0);
  useEffect(() => {
    document.title = `You clicked ${count} times`
  });

  return (
    <div>
      <p>You clicked {count}</p>
      <button onClick={()=>setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

Start with import useEffect from react, then use it like syntax beside. What will happen? useEffect will trigger if state or props is updated. When we click button “Click me”, it will updated count to count + 1.

When count updated, useEffect will trigger and execute code inside it, which is in this case, useEffect will execute to show how much we clicked the button “Click me”.

useEffect

If we feel that trigger every state and props updated is too much, you can tell React to *skip* applying an effect if certain values haven't changed between re-renders. To do so, pass an array as an optional second argument to useEffect:



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

In the example above, we pass [count] as the second argument. What does this mean? If the count is 5, and then our component re-renders with count still equal to 5, React will compare [5] from the previous render and [5] from the next render. Because all items in the array are the same (5 === 5), React would skip the effect. That's our optimization.

Other Hooks

- **useContext**, React Context is a way to manage state globally. It can be used together with the useState Hook to share state between deeply nested components more easily than with useState alone.
- **useRef**, allows you to persist values between renders. It can be used to store a mutable value that does not cause a re-render when updated. It can be used to access a DOM element directly.
- **useReducer**, It allows for custom state logic. If you find yourself keeping track of multiple pieces of state that rely on complex logic, useReducer may be useful.
- **useCallback**, returns a memoized callback function. This allows us to isolate resource intensive functions so that they will not automatically run on every render.
- **useMemo**, returns a memoized value. The useMemo Hook can be used to keep expensive, resource intensive functions from needlessly running.

Rules of Hooks

Hooks are JavaScript functions, but you need to follow two rules when using them.

Only Call Hooks at the Top Level

Don’t call Hooks inside loops, conditions, or nested functions. Instead, always use Hooks at the top level of your React function, before any early returns.

By following this rule, you ensure that Hooks are called in the same order each time a component renders. That’s what allows React to correctly preserve the state of Hooks between multiple useState and useEffect calls.

Only Call Hooks from React Functions

Don’t call Hooks from regular JavaScript functions. Instead, you can:

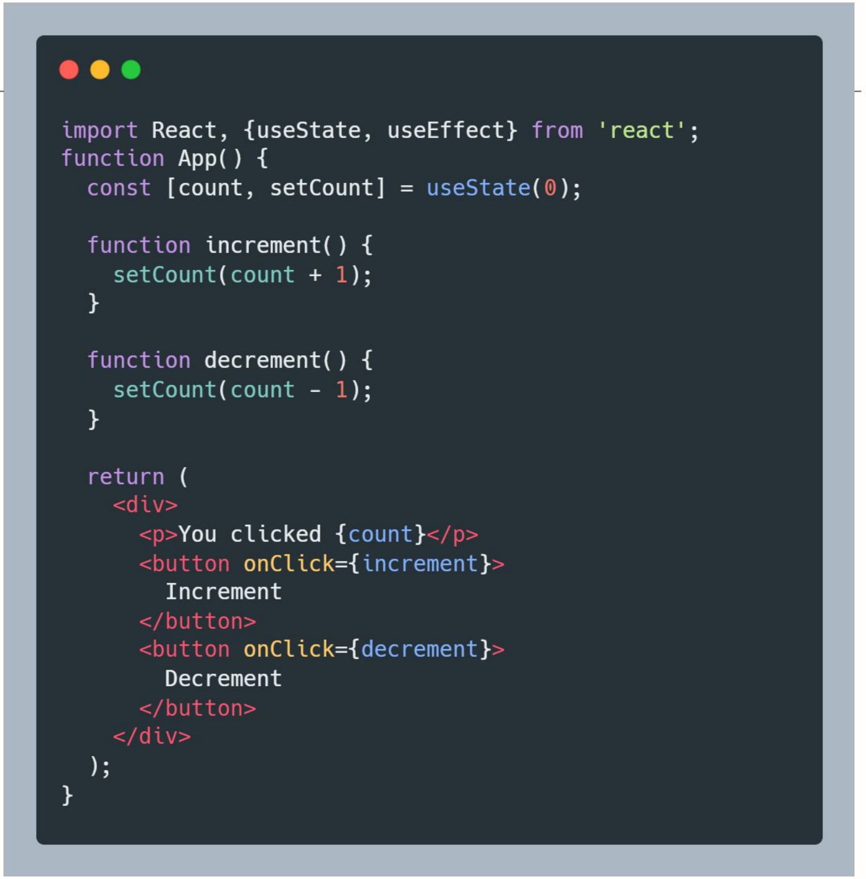
- ✅ Call Hooks from React function components.
- ✅ Call Hooks from custom Hooks.

By following this rule, you ensure that all stateful logic in a component is clearly visible from its source code.

Custom Hook

Building your own Hooks lets you extract component logic into reusable functions.

Now we will see an example of a counter use useState hook. In the beside example, we created a counter in App.js which is increments by 1 when we click on a increment button and decrements by 1 when we click on a decrement button.



```
import React, {useState, useEffect} from 'react';
function App() {
  const [count, setCount] = useState(0);

  function increment() {
    setCount(count + 1);
  }

  function decrement() {
    setCount(count - 1);
  }

  return (
    <div>
      <p>You clicked {count}</p>
      <button onClick={increment}>
        Increment
      </button>
      <button onClick={decrement}>
        Decrement
      </button>
    </div>
  );
}
```

Create your own hook

Suppose we need this counter in different places of our app in such cases we can build our custom react hook instead of creating same counter logic again and again.

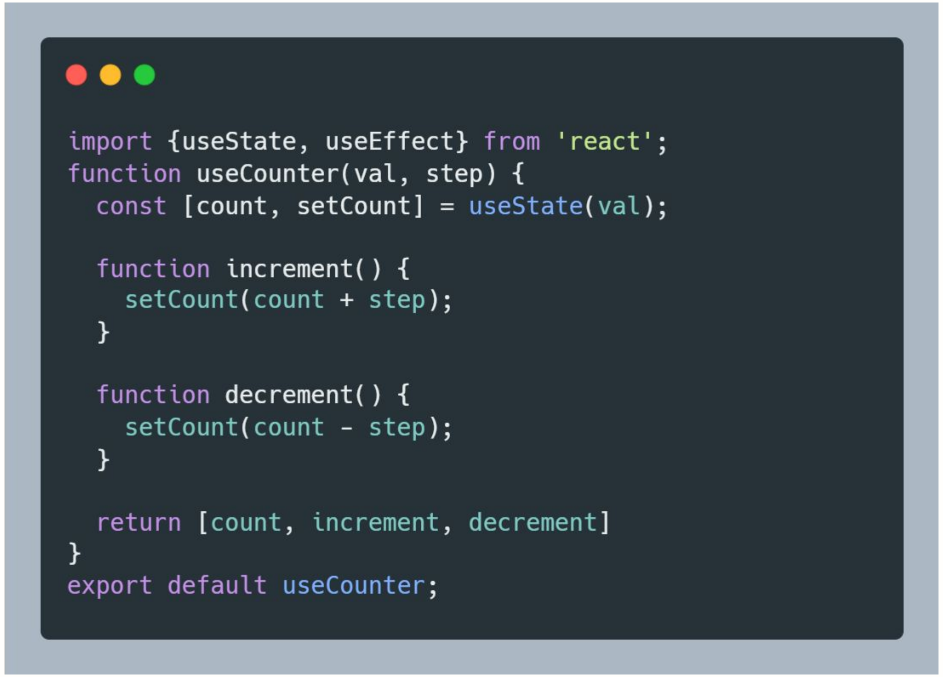
Custom hooks are JavaScript functions, whose name starts with use and they can call other react hooks.

Create your own hook

Now, we are removing the counter logic from the example before and creating our own custom hook called useCounter in counter-hook.js.

In the beside code, we created our own custom hook called useCounter with two parameters val and step and returning an array with count value , Increment and Decrement functions.

- **val**: Initial value of the counter.
- **step**: How many steps counter needs to increment or decrement.

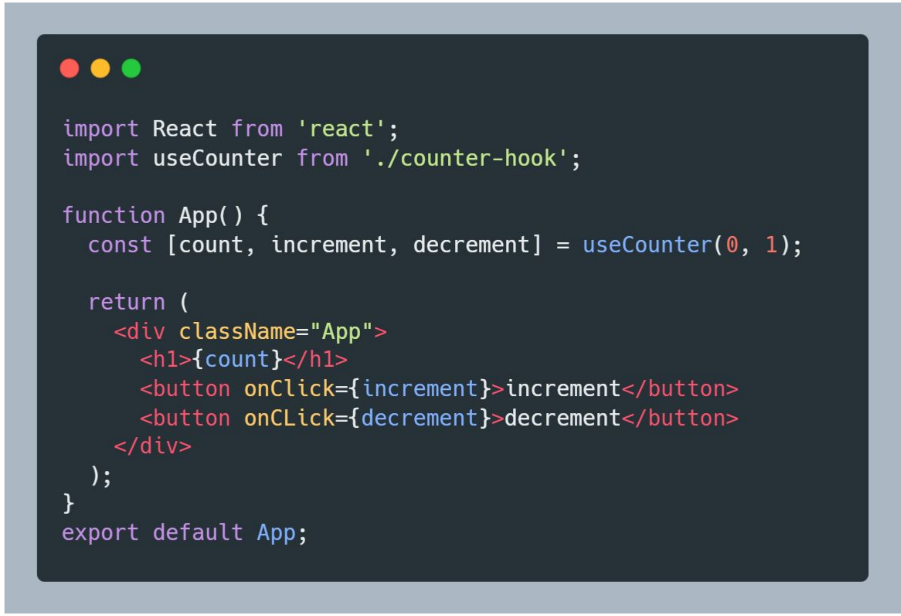


```
import {useState, useEffect} from 'react';
function useCounter(val, step) {
  const [count, setCount] = useState(val);

  function increment() {
    setCount(count + step);
  }

  function decrement() {
    setCount(count - step);
  }

  return [count, increment, decrement]
}
export default useCounter;
```

Create your own hook

Let's use the useCounter custom hook inside App.js component by importing it from the counter-hook.js file.

Now, we have a reusable counter logic we can use it whenever we need a counter in our react app.



```
import React from 'react';
import useCounter from './counter-hook';

function App() {
  const [count, increment, decrement] = useCounter(0, 1);

  return (
    <div className="App">
      <h1>{count}</h1>
      <button onClick={increment}>increment</button>
      <button onCLick={decrement}>decrement</button>
    </div>
  );
}
export default App;
```

Thank You!

Purwadhika
Digital Technology School