Cal State Fullerton

CPSC 349
React JS

By

Mahitha Pasupuleti

Handling Events

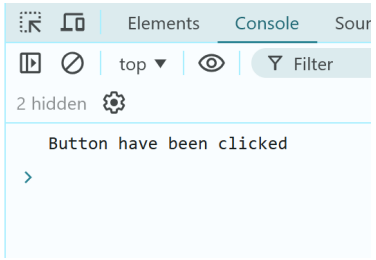
- You can respond to events by declaring *event handler* functions inside your components.
- Do not *call* the event handler function: you only need to *pass it down*. React will call your event handler when the user clicks the button.
- Notice how `onClick={clickme}` has no parentheses at the end!

```
function Button(){

    function clickme(){
        console.log(`Button have been clicked`)
    }
return(
<>
<button onClick={clickme}>Click Me</button>
</>);
}

export default Button;
```

Click Me

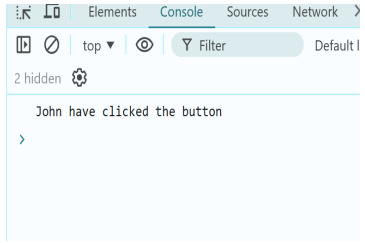



```
function Button(){

    function clickme(name){
        console.log(`${name} have clicked the button`)
    }
return(
<>
<button onClick={()=>clickme("John")}>Click Me</button>
</>);
}

export default Button;
```

Click Me



React Hooks

Functions starting with use are called *Hooks*.
You can either use the built-in Hooks or combine them to build your own.

Hooks allow function components to have access to state and other React features. Because of this, class components are generally no longer needed.

Some built-in hooks –

1. State hooks
   State lets a component “remember” information like user input. For example, a form component can use state to store the input value, while an image gallery component can use state to store the selected image index.

•useState declares a state variable that you can update directly.
•const [state, setState] = useState(initialState)
•Call useState at the top level of your component to declare one or more state variables.
•State cannot be passed to a component. We have props for that.

```jsx
import {useState} from 'react';
function Button(){

    const [cnt, setCnt]=useState(0);

    function clickme(){
        setCnt(cnt+1);
    }
return(
<>
Clicked on button {cnt} times!
<br />
<button onClick={clickme}>Click Me</button>
</>);
}

export default Button;
```

2. useEffect hook

- useEffect is a React Hook that lets you synchronize a component with an external system.
- The `useEffect` Hook allows you to perform side effects in your components.
- Some examples of side effects are: fetching data, directly updating the DOM, and timers.
- `useEffect` accepts two arguments. The second argument is optional.
- `useEffect(<function>, <dependency>)`

Effects **only run on the client.** They don't run during server rendering.

```
useEffect(() => {
  //Runs on every render
});
```

No dependency passed

```
useEffect(() => {
  //Runs only on the first render
}, []);
```

An Empty Array

```
useEffect(() => {
  //Runs on the first render
  //And any time any dependency value changes
}, [prop, state]);
```

Props or State value

- We have some more hooks, you can refer- https://react.dev/reference/react/hooks

Handling User Input

App.jsx | Form.jsx

components > Form.jsx > ...

```jsx
import { useState } from "react";

export default function Form() {
  const [firstname, setFirstname] = useState("");
  return (
    <div>
      <form>
        <input
          type="text"
          value={firstname}
          onChange={(e) => setFirstname(e.target.value)}
        />
      </form>
    </div>
  );
}
```

Multiple User Input

- Using state variables for all the inputs might prove difficult to handle. So, we set the state variable to an object. Making it easier to handle lots of value.
  - While updating the state variable, use the spread operator (...) to keep all the values as they were while changing the ones we need to update.

App.jsx Formmultiple.jsx

src > components > Formmultiple.jsx > Formmultiple

```jsx
import { useState } from "react";

export default function Formmultiple() {
  const [formObj, setFormObj] = useState({ fname: "", lname: "" });
  return (
    <div>
      <form>
        <label htmlFor="fname">Firstname</label>
        <input
          type="text"
          name="fname"
          value={formObj.fname}
          //   onChange={(e)=>setFormObj(e.target.value)}
        //   onChange={(e) => setFormObj({ fname: e.target.value, lname: "" })}
           onChange={(e) => setFormObj({ ...formObj, fname: e.target.value })}
        />
        <br />
        <label htmlFor="lname">Lastname</label>
        <input
          type="text"
          name="lname"
          value={formObj.lname}
        //   onChange={(e) => setFormObj({ lname: e.target.value, fname: "" })}
           onChange={(e) => setFormObj({ ...formObj, lname: e.target.value })}
        />
      </form>
    </div>
  );
}
```

Prop Drilling

- Passing data down the component tree through props.

- When a child component needs data from a parent component, that data is explicitly passed down as props from the parent to the child. This process can be repeated for components further down the hierarchy, essentially "drilling" the props down the tree.

- In extreme cases, components become overly reliant on receiving specific props from their parents, leading to a situation known as "prop hell," where changes in one part of the application require changes to propagate through many components.

Lifting the states

- Moving shared state to a common ancestor component in the hierarchy.
- Instead of drilling data down through props, you identify the closest common ancestor component that needs access to the data. This ancestor component manages the state, and then both child components receive the data as props from this shared parent.
- Benefits:
    - Components become more focused on specific tasks and less reliant on props passed down from multiple levels.
    - Updates to shared data are centralized in the common ancestor component.
    - Since the state resides in a single place, consistency becomes simpler

Reference

- https://react.dev/reference/react/hooks
- https://www.w3schools.com/react/react_hooks.asp