Front End III

useToggle

Básicamente, lo que hace este Hook es permitirnos almacenar y alternar un estado entre un valor verdadero o falso. Es útil cuando queremos convertir una acción en su acción opuesta, por ejemplo: mostrar u ocultar un modal, mostrar más o menos texto, abrir o cerrar un menú.

Escribamos un pequeño ejemplo de un botón que cambia el estado de ON a OFF y viceversa, primero sin un Hook personalizado. Utilizaremos **useState** para definir una variable de estado llamada **state** que controlará el estado del botón:

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

Luego, crearemos una función llamada **setToggle** que utilizará el Hook **useCallback** de React para invertir el estado del botón dependiendo de su estado anterior:

```
  const setToggle = useCallback(
      () => setState(state => !state),
      [state]
  );
```

En definitiva, el componente **App** lucirá así:

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

const App = () => {
  const [state, setState] = useState(false);
```

```
    const setToggle = useCallback(
        () => setState(state => !state),
        [state]
    );

    return (
        <button onClick={setToggle}>
            { state ? 'ON' : 'OFF' }
        </button>
    );

};

export default App;
```

Pero, ¿qué sucede si quisiéramos tener más de un botón en nuestra aplicación? De la manera en que hemos implementado el mismo hasta ahora, deberíamos duplicar toda la lógica en el componente **App**. Es decir, tendríamos dos estados y dos funciones para actualizarlo. Si bien esto funciona perfectamente, estamos generando una repetición innecesaria de código, ya que en ambos casos el funcionamiento es idéntico. Para resolver esto, podemos valernos del uso de los Hooks personalizados. Veamos entonces cómo podemos encapsular y reutilizar el código del toggle basándonos en lo que aprendimos.

Lo que haremos es crear un archivo con el nombre del Hook (**useToggle.jsx**), y mover todo lo relacionado con los Hooks **useState** y **useCallback** a ese archivo, retornando tanto la variable de estado (**state**) como la función que lo controla (**toggleState**):

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

```
export function useToggle(initialState = false) {

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

    const toggleState = useCallback(
        () => setState(state => !state),
        [state]
    );

    return [state, toggleState];
}
```

Ahora, crearemos un componente llamado **ToggleButton** que va a emplear nuestro Hook personalizado **useToggle** de esta manera:

```
    const [toggle, setToggle] = useToggle();

    return (
        <button onClick={setToggle}>
            { toggle ? 'ON' : 'OFF' }
        </button>
    );
```

El componente **ToggleButton** luce así:

```
import React from 'react';
import { useToggle } from "./useToggle.jsx";

export function ToggleButton() {
```

```
    const [toggle, setToggle] = useToggle();

    return (
        <button onClick={setToggle}>
            { toggle ? 'ON' : 'OFF' }
        </button>
    );
}
```

Por último, creamos un componente **App** que renderiza dos botones de tipo **ToggleButton**:

```
import React from 'react';
import { ToggleButton } from "./ToggleButton.jsx";
import "./app.scss";

const App = () => {
    return (
        <>
            <ToggleButton/>
            <ToggleButton/>
        </>
    );
};

export default App;
```

De esta manera escribimos la lógica una única vez y luego podemos crear tantos botones como deseamos. Asimismo, basándonos en lo que aprendimos anteriormente, podemos estar seguros de que cada componente **ToggleButton** manejará su propio estado ON/OFF en forma separada, lo que nos permitirá, por ejemplo, que un botón se encuentre encendido y el otro apagado al mismo tiempo.

Pero eso no es todo. Si en el futuro deseamos crear otro tipo de componente que emplee un estado con dos posibilidades (verdadero o falso), también podremos utilizar nuestro Hook **useToggle** sin necesidad de volver a escribir esta lógica dentro del nuevo componente.