Hooks II

Como conocemos los Hooks son una nueva API de la librería de React que nos permite tener estado, y otras características de React, en los componentes creados con una function. Esto, antes, no era posible y nos obligaba a crear un componente con class para poder acceder a todas las posibilidades de la librería. Y de ahí viene el nombre. Hooks es gancho y, precisamente, lo que hacen, es que te permiten enganchar tus componentes funcionales a todas las características que ofrece React.

useReducer

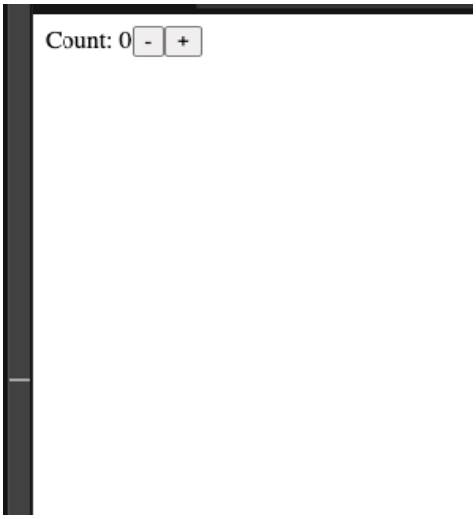
Una alternativa a useState. Acepta un reducer de tipo (state, action) => newState y devuelve el estado actual emparejado con un método dispatch. (Si está familiarizado con *Redux*, ya sabe cómo funciona).

useReducer a menudo es preferible a useState cuando se tiene una lógica compleja que involucra múltiples subvalores o cuando el próximo estado depende del anterior. useReducer además te permite optimizar el rendimiento para componentes que activan actualizaciones profundas, porque puedes pasar hacia abajo dispatch en lugar de *callbacks*.

Ejemplo del uso en un contador:

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

// Estado Inicial      1
const initialState = { count: 0 };

// Reductor      2
const reducer = (state, action) => {
  switch (action.type) {
    case "increment":
      return { count: state.count + 1 };
    case "decrement":
      return { count: state.count - 1 };
    default:
      throw new Error();
  }
};

const App = () => {
  // Uso del hook  3
  const [state, dispatch] = useReducer(reducer, initialState);

  return (
    <>
      Count: {state.count}
      <button onClick={() => dispatch({ type: "decrement" })}>-</button>
      <button onClick={() => dispatch({ type: "increment" })}>+</button>
    </>
  );
};

export default App;
```

Render



Como podemos observar en la siguiente imagen con el hook useReducer consta de 3 principales partes:
1 – Un estado inicial o Initial State.
2 – Un reductor o reducer.
3 – El uso del propio hook .

Estado Incial o Initial State: Es el estado o valor con el cual va a inicializara nuestro Hook, por ende cuando lo usemos por primera vez va a tener un valor por defecto.

Reducer: El reducer o reductor es una función que recibe como parámetros dos valores un State(Estado) y una Action (acción) y lo que contiene dicha función es un Switch que va a discriminar su funcionalidad en base a la acción que nosotros le mandemos como parámetro. Para que luego ejecute su lógica internamente, en el ejemplo de arriba podemos ver que el botón que contiene un +, lo que hace es incrementar el valor del contador (state.count) en + 1.

¡ Recuerden que siempre que utilizan un Switch deben colocar un caso por default !

El uso del Hook: Para el uso del hook obtenemos el State que va a ser el valor del estado y el dispatch, el cual va a ejecutar nuestra lógica, y como parámetro vamos a enviarle el reducer y su estado inicial.
Para luego utilizarlo de la siguiente manera:

```
Count: {state.count}
<button onClick={() => dispatch({ type: "decrement" })}>-</button>
```

Count va a renderizar el estado actual de nuestro hook y el botón va a ejecutar la función del reducer decrement

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

Por ende la lógica que va a ejecutar este botón es la de restar – 1 el valor de nuestro estado.

useID

Es un Hook para generar ID únicos que son estables en el servidor y el cliente, al tiempo que evita los desajustes de hidratación.

Un básico ejemplo de su uso:

```
const Checkbox = () => {
  const id = useId();
  return (
    <>
      <label htmlFor={id}>Te gusta React?</label>
      <input id={id} type="checkbox" name="react"/>
    </>
  );
};
```

Nota : useId no es para generar claves en una lista. Las claves deben generarse a partir de sus datos.

useCallback

Devuelve un callback memorizado.

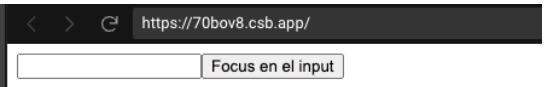
Pasa un callback en línea y un arreglo de dependencias (Como en el useEffect) . useCallback devolverá una versión memorizada del callback que solo cambia si una de las dependencias ha cambiado. Esto es útil cuando se transfieren callbacks a componentes hijos optimizados que dependen de la igualdad de referencia para evitar renders innecesarias (por ejemplo, shouldComponentUpdate).

```
const SumarNumeros = ( a , b ) => a + b

const memoizedCallback = useCallback(
  () => {
    SumarNumeros(numero1, numero2);
  },
  [numero1, numero2],
);
```

useRef

Devuelve un objeto *ref* mutable cuya propiedad .current se inicializa con el argumento pasado (initialValue). El objeto devuelto se mantendrá persistente durante la vida completa del componente.

```
import { useRef } from "react";

const App = () => {
  const inputEl = useRef(null);
  const onButtonClick = () => {
    // `current` apunta al elemento de entrada en este caso al Input
    inputEl.current.focus();
  };
  return (
    <>
      <input ref={inputEl} type="text" />
      <button onClick={onButtonClick}>Focus en el input</button>
    </>
  );
};

export default App;
```



Por ende en el siguiente ejemplo lo que va a realizar el botón es hacer un Focus en el input de su izquierda.

En esencia, useRef es como una "caja" que puedes mantener en una variable mutable en su propiedad .current.

Puede que estes familiarizado con las referencias principalmente como un medio para acceder al DOM. Si pasas un objeto de referencia a React con , React configurará su propiedad .current al nodo del DOM correspondiente cuando sea que el nodo cambie.

Sin embargo, useRef es útil para más que el atributo ref. Es conveniente para mantener cualquier valor mutable que es similar a como usarías campos de instancia en las clases.

Ten en cuenta que useRef *no* notifica cuando su contenido cambia. Mutar la propiedad .current no causa otro renderizado.