UNIVERSIDAD MARIANO GALVEZ DE GUATEMALA

Facultad Ingeniería en Sistemas
CURSO: Desarrollo Web
CATEDRÁTICO: Ing. Carmelo Estuardo Mayen Monterroso
SEMESTRE: Octavo

ALUMNO:
José Daniel Daniel Bran Benito **1790-22-15044**

Tarea:

INVESTIGACION HOOK EN REACT

Chiquimulilla, Santa Rosa, 19 de Septiembre del 2025

Hook useState en React

Firma : const [estado, setEstado] = useState(valorInicial);

El Hook **useState** es un Hook que permite añadir el estado de React a un componente funcional y sirve para manejar el estado de los elementos de un componente de manera que permite actualizar el estado de una variable y solo renderizar el tag html donde se lo use, de manera que cuando el estado cambia el componente responde volviendo a renderizar solo la parte del código afectada por la variable de estado mantenida por el hook **useState**.

Debes usar el Hook useState en un componente funcional de React siempre que necesites almacenar y gestionar un valor que pueda cambiar con el tiempo y que afecte directamente a lo que el usuario ve en la interfaz (UI), como contadores, datos de formularios, estados de visibilidad o el estado de un usuario.

Ejemplo equivalente en forma de clase

Si has usado clases en React previamente, este código te resultará familiar:

```
class Example extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0
    };
  }

  render() {
    return (
      <div>
        <p>You clicked {this.state.count} times</p>
        <button onClick={() => this.setState({ count: this.state.count + 1
})}>
          Click me
        </button>
      </div>
    );
  }
}
```

El estado empieza como { count:0 } y se incrementa state.count cuando el usuario hace click a un botón llamando a this.setState(). Usaremos fragmentos de esta clase en toda la página.

Hook useEffect en React

Firma: La firma de useEffect en React consiste en llamar al hook con dos argumentos: el primero es una función que contiene la lógica del efecto secundario (que puede ser una función de limpieza), y el segundo es un array de dependencias que determina cuándo se vuelve a ejecutar ese efecto.

El gancho useEffect de React permite ejecutar efectos secundarios en componentes de función. Reemplaza métodos de ciclo de vida como componentDidMount, componentDidUpdate y componentWillUnmount en componentes de clase.

El hook **useEffect** de **React** es una herramienta poderosa que nos permite controlar los efectos secundarios en nuestros componentes. Los efectos secundarios son cualquier cambio de estado que no sea el resultado directo de una actualización de estado. Algunos ejemplos comunes de efectos secundarios incluyen:

- Realizar una petición HTTP para obtener datos de una API
- Escuchar eventos del teclado o del ratón
- Cambiar el título de la página

El hook *useEffect* nos permite controlar estos efectos secundarios de forma declarativa, lo que significa que podemos especificar exactamente cuándo deben ocurrir.

Un ejemplo de uso básico de *useEffect* podría ser el siguiente:

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


const MiComponente = () => {
  const [data, setData] = useState(null);


  useEffect(() => {
    fetch('https://api.example.com/data')
      .then(response => response.json())
      .then(data => setData(data));
  }, []);


  return (
    <div>
```

```
      {data ? (
        <div>{data.title}</div>
      ) : (
        <div>Cargando...</div>
      )}
    </div>
  );
}
```

useMemo

Firma: La firma del hook useMemo en React es useMemo(calcularValor, dependencias). Recibe una función que devuelve el valor a memoizar y un array de dependencias. El valor solo se recalcula cuando una de las dependencias cambia; de lo contrario, se devuelve el valor memorizado.

El gancho useMemose utiliza en React para memorizar el resultado de una función costosa de calcular, evitando que se ejecute en cada renderizado del componente. Es útil especialmente cuando tienes operaciones computacionales intensas o componentes que se renderizan frecuentemente con las mismas props.

Ejemplo 1: Cálculo de un valor costoso

Supongamos que tenemos una función que calcula el factorial de un número, y queremos evitar que esta función se ejecute en cada renderizado.

```
importar  React , { useMemo } de  'react' ;

función factorial(n) {
  si (n < 0) devuelve 0;
  si (n === 0) devuelve 1;
  deja resultado = 1;
  para (deja i = 1; i <= n; i++) {
    resultado *= i;
  }
  devuelve resultado;
}const MiComponente = ({ num }) => {
  const factorialResult = useMemo(() => {
    return factorial(num);
  }, [num]); // La función sólo se volverá a calcular si 'num' cambia  return (
    <div>
```

```
    El factorial de {num} es: {factorialResult}
  </div>
 );
};exportar predeterminado MyComponent;
```

Hook useRef

El useRef()Hook es una función integrada de React que conserva los valores entre renderizaciones de componentes. A diferencia de las variables de estado gestionadas por useState, los valores almacenados en un objeto de referencia permanecen inalterados entre renderizaciones, lo que lo hace ideal para escenarios donde los datos no afectan directamente a la interfaz de usuario, pero son esenciales para el comportamiento del componente.

Casos de uso del gancho useRef() de React

Más allá de su capacidad para persistir valores, el useRef()Hook tiene varias funciones vitales en las aplicaciones React:

Acceder a elementos DOM

Imagine una página de inicio de sesión donde los usuarios deben ingresar su nombre de usuario y contraseña. Para mejorar la experiencia del usuario, puede dirigir automáticamente su atención al campo de nombre de usuario al cargar la página.

Para lograr esto, utilice la useRefcapacidad de acceder a los elementos DOM renderizados. Esta función devuelve el elemento DOM referenciado y sus propiedades, lo que permite manipulaciones directas.

```
importar { useRef , useEffect } de “react”

   función Iniciar sesión ( ) {

    const usernameRef = useRef ( null )


    usarEfecto ( ( )) => {

      usernameRef . current . focus ( )

    } , [ ] )

   devolver (

    < >

     < formulario >

       < tipo de entrada = "texto" ref = { usernameRef } / >
```

```
      </form>
     < / >
   )
  }
```

Hook useContext

useContext es un Hookde React que permite a los componentes funcionales acceder al valor de un contexto proporcionado por el componente Proveedor más cercano en la jerarquía de componentes.

useContext toma como argumento el objeto del contexto creado con React.createContext y devuelve el valor actual del contexto.

La sintaxis básica y pasos para utilizar el Hook useContext es la siguiente:

1- Crear nuestro contexto.

```
import React from 'react';

export const myContext = React.createContext('Default value');
```

El valor inicial para el contexto myContext es 'Default value'. Si algún componente Consumidor intenta consumir este contexto, pero no encuentra un Proveedor correspondiente en su jerarquía, se utilizará este valor 'Default value' como el valor del contexto.

2- Crear nuestro proveedor

```
import React from "react";
import { myContext } from "./context";

export function ProviderContextComponent() {
  const sharedData = "My shared data!";

  return (
    <myContext.Provider value={sharedData}>

      {/* children components */}

    </myContext.Provider>
  );
}
```

El componente ProviderContextComponent actúa como el proveedor y envuelve a los componentes hijos con el contexto proporcionado por Context.Provider , a su vez, estamos sobreescribiendo el valor inicial por defecto. Por último debemos agregar en el apartado {/* children components } los componentes consumidores para obtener el valor compartido.

3- Crear nuestro consumidor

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

export function ConsumerContextComponent() {
  const data = useContext(myContext);

  return (
    <div>Data shared is: {data}</div>
  );
};
```

El Hook useContext se obtiene importando la función useContext desde la librería de React directamente. Recordar que solo estará disponible en versiones de 16.8 o superior.

Hook useReducer

A useReduceres un gancho en React que permite agregar un reducera un componente. Toma la función reductora y un initialStatecomo argumentos. useReducerTambién devuelve un array con la función actual statey una dispatchfunción.

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

Familiaricémonos con lo que significan los parámetros:

- state: representa el valor actual y se establece en el initialStatevalor durante la representación inicial.
- dispatch: es una función que actualiza el valor del estado y siempre activa una nueva representación, al igual que la función de actualización en useState.
- reducer: es una función que contiene toda la lógica de actualización del estado. Toma el estado y la acción como argumentos y devuelve el siguiente estado.
- initialState:alberga el valor inicial y puede ser de cualquier tipo.

Ejemplo de reductor en el mundo real

Imagínese a un despachador que trabaja para una empresa de compras en línea que va a un almacén para recoger los productos o artículos que luego distribuirá a las personas que los pidieron.

El despachador se identifica y reclama la mercancía para su envío al gerente de almacén. El gerente se acerca a la caja que contiene los pedidos enviados y localiza la mercancía que se le entregará. También inicia sesión en el sistema de inventario y realiza las evaluaciones antes de entregar la mercancía al despachador.

Este escenario también se puede traducir como:

- El despachador realiza una solicitud de actualización o activa un proceso como la dispatchfunción.
- El despachador realiza una acción de 'reclamación de mercancías' como el despacho actioncon una typepropiedad.
- El jefe de almacén realiza la clasificación y actualización necesarias al igual que la reducerfunción.
- La caja que contiene toda la mercancía se actualiza según la cantidad despachada. Esto funciona como una stateactualización.

Espero que este escenario del mundo real le aclare todo el proceso.

Eche un vistazo al código completo una vez más y asimile el proceso.

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


function reducer(state, action) {
  console.log(state, action);
  switch (action.type) {
    case "increment":
      return { ...state, count: state.count + 1 };
    case "decrement":
      return { ...state, count: state.count - 1 };
    default:
      return "Unrecognized command";
  }
}
const initialState = { count: 0 };
export default function App() {
  const [state, dispatch] = useReducer(reducer, initialState);
```

```
  function handleIncrement() {
    dispatch({ type: "increment" });
  }
  function handleDecrement() {
    dispatch({ type: "decrement" });
  }
  return (
    <>
      <h1>Count:{state.count}</h1>
      <button onClick={handleIncrement}>Increment</button>
      <button onClick={handleDecrement}>Decrement</button>
    </>
  );
}
```

Hook useCallback

El hook useCallback es un hook que nos permite memorizar una función. Esto quiere decir que si la función que le pasamos como parámetro no ha cambiado, no se ejecuta de nuevo y se devuelve la función que ya se había calculado.

```
import { useCallback } from 'react'


function Counter({ count, onIncrement }) {
  const handleIncrement = useCallback(() => {
    onIncrement(count)
  }, [count, onIncrement])


  return (
    <div>
```

```
      <p>Contador: {count}</p>
      <button onClick={handleIncrement}>Incrementar</button>
    </div>
  )
}
```

En este caso, el componente Counter recibe una prop count que es un número y una prop onIncrement que es una función que se ejecuta cuando se pulsa el botón.

El hook useCallback recibe dos parámetros: una función y un array de dependencias. La función se ejecuta cuando el componente se renderiza por primera vez y cuando alguna de las dependencias cambia, en este ejemplo la prop count o la prop onIncrement.

La ventaja es que si la prop count o la prop onIncrement no cambian, se evita la creación de una nueva función y se devuelve la función que ya se había calculado previamente.