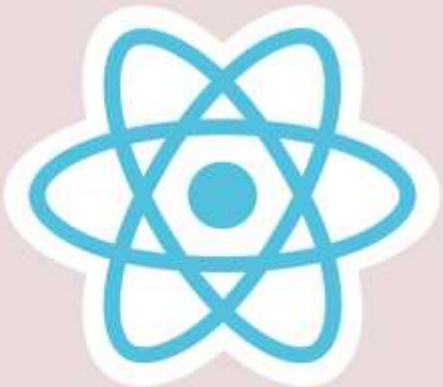
5_1.2

www.opentowork.es

ReactJS

Hooks

Los Hooks en ReactJS son una adición poderosa que permite a los componentes funcionales tener características que antes solo estaban disponibles en los componentes de clase, como el manejo del estado y el acceso al ciclo de vida del componente.



#01/24@ mihifidem

Tema 4: Hooks

6.Hook useReducer

6.1. Cómo Funciona useReducer

6.2. Ejemplo Básico

6.3. Ventajas de useReducer

6.4.Conclusión

7.Hook useCallback

7.1. Cómo Funciona useCallback

7.2. Ejemplo de Uso

7.3. Ventajas de useCallback

7.4. Conclusión

8.Hook useMemo

8.1. Cómo Funciona useMemo

8.2. Ejemplo de Uso

8.3 Ventajas de useMemo

8.4. Conclusión

1. Introducción

1.1 ¿Qué es un hook?

Un hook en ReactJS es una función especial que permite a los componentes funcionales "engancharse" a características de React como el estado y el ciclo de vida, que anteriormente solo estaban disponibles en componentes de clase. Los hooks fueron introducidos en React con la versión 16.8 para simplificar la escritura de componentes y para reutilizar la lógica de estado y efectos entre ellos sin cambiar la jerarquía de componentes.

Los hooks más comunes incluyen:

1. **useState**: Permite a los componentes funcionales tener su propio estado.
2. **useEffect**: Para ejecutar efectos secundarios (como solicitudes de datos, suscripciones, o actualizaciones del DOM) en el componente.
3. **useContext**: Facilita el acceso y la gestión del contexto, proporcionando una manera de compartir valores entre diferentes componentes sin tener que pasar explícitamente props a través de cada nivel del árbol de componentes.
4. **useReducer**: Ofrece una alternativa más robusta a **useState** para manejar estados complejos en componentes.

Los hooks aportan flexibilidad y reutilización de código, permitiendo a los desarrolladores escribir componentes más limpios y mantenibles. Además, hacen que la lógica relacionada con el estado y los efectos sea más fácil de organizar y entender en los componentes funcionales.

Ejemplo básico que ilustra el uso de algunos hooks comunes en ReactJS, como useState y useEffect. Consideremos un componente funcional simple que muestra un contador y tiene un botón para incrementarlo:

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

  function Counter() {
      // useState para manejar el estado del contador
      const [count, setCount] = useState(0);

      // useEffect para realizar alguna acción cuando el componente se
monta o el estado cambia
      useEffect(() => {
          document.title = `Has clickeado ${count} veces`;
      }, [count]); // Este efecto se ejecuta cada vez que 'count' cambia

      return (
          <div>
              <p>Has clickeado {count} veces</p>
              <button onClick={() => setCount(count + 1)}>
                  Clickea aquí
              </button>
          </div>
      );
  }

  export default Counter;
```

En este ejemplo:

You must **import** Hooks from **react**.

El hook useState se utiliza para crear una variable de estado count con un valor inicial de 0. La función setCount se usa para actualizar este valor.

El hook useEffect se emplea para cambiar el título del documento cada vez que el valor de count cambia. La dependencia [count] indica que el efecto se debe ejecutar solo cuando el valor de count se actualiza.

El botón en el JSX tiene un onClick handler que llama a setCount(count + 1) para incrementar el valor de count cada vez que se hace clic en él.

Este es un ejemplo simple pero ilustrativo del uso de hooks en React para manejar estados y efectos en componentes funcionales.

1.2. Reglas de los hooks

Las reglas de los ganchos (hooks) en React son fundamentales para asegurar que su uso sea consistente y no cause problemas inesperados en las aplicaciones. Aquí detallo las tres reglas que mencionaste:

1. **Los hooks solo se pueden llamar dentro de los componentes de función de React**: Esto significa que no puedes usar hooks en componentes de clase o funciones regulares de JavaScript. Los hooks están diseñados para trabajar con los componentes funcionales de React, aprovechando sus características y ciclo de vida.

2. **Los hooks solo se pueden llamar en el nivel superior de un componente**: Los hooks deben ser llamados en el nivel superior de un componente de React, no dentro de bucles, condiciones, o funciones anidadas. Esto es crucial para que React mantenga el orden de los hooks entre múltiples renderizados. Si los hooks se llaman en un orden diferente en cada renderizado, React no podría rastrear correctamente el estado y los efectos secundarios asociados con cada hook.

3. **Los hooks no pueden ser condicionales**: No se deben usar hooks dentro de sentencias condicionales (if, for, while, etc.), ya que violaría la regla del orden de llamada. Todos los hooks deben ser utilizados en el mismo orden en cada renderización. Permitir hooks condicionales podría llevar a que diferentes renderizados llamen a diferentes hooks, lo que rompería la consistencia en el seguimiento del estado y de los efectos por parte de React.

Estas reglas aseguran que el uso de hooks sea seguro y predecible, permitiendo a React gestionar adecuadamente el estado y el ciclo de vida de los componentes. Incumplir estas reglas puede llevar a errores difíciles de rastrear y comportamientos inesperados en las aplicaciones React.

2. Hook useState

El hook **useState** en React es una de las características introducidas en la versión 16.8 que permite a los componentes funcionales tener un estado interno. Antes de los hooks, los estados solo podían ser usados en componentes de clase. **useState** es una forma sencilla y elegante de añadir estado a componentes funcionales.

2.1. Cómo Funciona useState

1. Para usar el hook useState,primero debemos importarlo a nuestro componente.

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

2. **Inicialización**: Se invoca **useState** con un valor inicial para el estado. Este valor puede ser de cualquier tipo: un número, una cadena, un booleano, un objeto, un array, etc.

```
const [value, setValue] = useState(initialValue);
```

En esta línea, **useState** devuelve un par de valores: el estado actual (**value**) y una función que permite actualizar ese estado (**setValue**).

Ejemplo

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

function FavoriteColor() {
  const [color, setColor] = useState("");
}
```

Observe que nuevamente, estamos desestructurando los valores devueltos por useState.

El primer valor, el color, es nuestro estado actual.

El segundo valor, setColor, es la función que se utiliza para actualizar nuestro estado.

Estos nombres son variables que pueden denominarse como desee.

Por último, establecemos el estado inicial en una cadena vacía: useState("")

2.2. Ejemplo Básico

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

function Counter() {
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>El contador está a {count}</p>
      <button onClick={() => setCount(count + 1)}>
        Incrementar
      </button>
    </div>
  );
}
```

En este ejemplo, **useState** se utiliza para crear un contador. El estado inicial es 0. Cada vez que se hace clic en el botón, se llama a **setCount** con el nuevo valor del contador, que incrementa el contador en 1. Esto provoca que el componente **Counter** se vuelva a renderizar con el nuevo valor de **count**.

2.3. Ventajas de useState

- **Simplicidad y Claridad**: Hace que el código sea más legible y fácil de entender.
- **Menos Código Boilplate**: Reduce la cantidad de código necesario para manejar estados en comparación con los componentes de clase.
- **Facilita la Reutilización del Estado**: Puedes extraer la lógica del estado en una función personalizada y reutilizarla en varios componentes.

2.4. Lectura del Estado

Puedes usar la variable de estado (**value** en el ejemplo anterior) en tu componente para mostrar su valor en la UI, realizar cálculos, etc.

```
import { useState } from "react";
import ReactDOM from "react-dom/client";

function FavoriteColor() {
  const [color, setColor] = useState("red");

  return <h1>My favorite color is {color}!</h1>
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<FavoriteColor />);
```

Tema 4: Hooks

React es un excelente ejemplo de cómo utilizar el hook **useState** para manejar el estado dentro de un componente funcional. Vamos a desglosarlo:

1. **Importación de useState y ReactDOM**:
   - Primero, importas **useState** de **react**, lo cual es necesario para usar este hook en tu componente.
   - También importas **ReactDOM** para poder montar tu componente en el DOM.

2. **Componente FavoriteColor**:
   - Creas un componente funcional llamado **FavoriteColor**.
   - Dentro del componente, utilizas **useState** para crear un estado llamado **color**, con un valor inicial de **"red"**. La función **setColor** se utilizaría para cambiar este estado, aunque en este ejemplo particular no se está cambiando el color.
   - El componente devuelve un elemento **<h1>** que muestra el color favorito actual.

3. **Renderizando el Componente en el DOM**:
- Creas una raíz de React usando **ReactDOM.createRoot** y le pasas el elemento del DOM donde quieres montar tu aplicación React. En este caso, es un elemento con **id='root'**.
- Finalmente, renderizas tu componente **FavoriteColor** dentro de esta raíz.

Este código, al ejecutarse, mostrará un título **<h1>** que dice "My favorite color is red!" en la página. Si quieres cambiar el color, podrías añadir un botón y usar **setColor** para actualizar el estado. Por ejemplo:

```
function FavoriteColor() {
  const [color, setColor] = useState("red");

  return (
    <div>
      <h1>My favorite color is {color}!</h1>
      <button onClick={() => setColor("blue")}>
        Cambiar a Azul
      </button>
    </div>
  );
}
```

En este ejemplo extendido, al hacer clic en el botón, el color cambiará a azul y el componente se volverá a renderizar para reflejar este cambio.

2.5. Actualización del Estado:

Para actualizar el estado, usas la función proporcionada por **useState** (**setValue**). Cuando esta función se llama, el componente se vuelve a renderizar con el nuevo valor del estado.

```
setValue(newValue);
```

El modificado de React ahora incluye un botón que permite actualizar el estado del color. Este es un buen ejemplo para ilustrar cómo se puede cambiar el estado en React utilizando el hook useState. Aquí está el análisis de tu código:

1. **Uso de useState:**
   - const [color, setColor] = useState("red"); inicializa el estado color con el valor "red" y setColor es la función que se utilizará para actualizar este estado.
2. **Renderizado del Componente:**
   - El componente FavoriteColor muestra el color actual en un elemento <h1>.
   - Hay un botón que, al hacer clic, cambiará el estado color a "blue" mediante la función setColor.
3. **Actualización de Estado:**
   - La línea onClick={() => setColor("blue")} en el botón es crucial. Define un manejador de eventos onClick que llama a setColor con el nuevo color "blue".
   - Importante: Estás utilizando setColor para actualizar el estado, en lugar de modificar el estado directamente (como color = "blue"), lo cual es la práctica correcta en React. React entonces re-renderizará el componente FavoriteColor con el nuevo valor del estado.
4. **Montaje del Componente en el DOM:**
   - Usas ReactDOM.createRoot(document.getElementById('root')) para crear una raíz y montar tu componente FavoriteColor en el elemento con id='root' del DOM.

Tema 4: Hooks

Este código muestra cómo manejar el estado de manera efectiva en React. Al hacer clic en el botón, React actualizará el estado color y volverá a renderizar el componente para reflejar el nuevo estado, cambiando el texto a "My favorite color is blue!".

```
import { useState } from "react";
import ReactDOM from "react-dom/client";

function FavoriteColor() {
 const [color, setColor] = useState("red");

 return (
  <>
   <h1>My favorite color is {color}!</h1>
   <button
    type="button"
    onClick={() => setColor("blue")}
   >Blue</button>
  </>
 )
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<FavoriteColor />);
```

2.6. ¿Qué puede tener el estado?

Los useState Hook se puede usar para realizar un seguimiento de cadenas, números, booleanos, matrices, objetos y cualquier combinación de estos!

Podríamos crear múltiples ganchos de estado para rastrear valores individuales.

Ejemplo: Crear múltiples ganchos de estado:

```
import { useState } from "react";
import ReactDOM from "react-dom/client";

function Car() {
  const [brand, setBrand] = useState("Ford");
  const [model, setModel] = useState("Mustang");
  const [year, setYear] = useState("1964");
  const [color, setColor] = useState("red");

  return (
    <>
      <h1>My {brand}</h1>
      <p>
        It is a {color} {model} from {year}.
      </p>
    </>
  )
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Car />);
```

¡O simplemente podemos usar un estado e incluir un objeto!

Tema 4: Hooks

Ejemplo: Crea un solo gancho que contenga un objeto:

```
import { useState } from "react";
import ReactDOM from "react-dom/client";

function Car() {
  const [car, setCar] = useState({
    brand: "Ford",
    model: "Mustang",
    year: "1964",
    color: "red"
  });

  return (
    <>
      <h1>My {car.brand}</h1>
      <p>
        It is a {car.color} {car.model} from {car.year}.
      </p>
    </>
  )
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Car />);
```

2.7. Actualización de objetos y matrices en el estado

Cuando se actualiza el estado, se sobrescribe todo el estado.

¿Qué pasa si solo queremos actualizar el color de nuestro automóvil?

Si solo llamáramos setCar({color: "blue"}), esto eliminaría la marca, el modelo y el año de nuestro estado.

Podemos usar el operador de spread de JavaScript para ayudarnos.

Ejemplo:
Use el operador de extensión de JavaScript para actualizar solo el color del automóvil:

```
import { useState } from "react";
import ReactDOM from "react-dom/client";

function Car() {
 const [car, setCar] = useState({
  brand: "Ford",
  model: "Mustang",
  year: "1964",
  color: "red"
 });

 const updateColor = () => {
  setCar(previousState => {
   return { ...previousState, color: "blue" }
  });
 }

 return (
  <>
   <h1>My {car.brand}</h1>
   <p>
    It is a {car.color} {car.model} from {car.year}.
   </p>
   <button
    type="button"
    onClick={updateColor}
   >Blue</button>
  </>
 )
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Car />);
```

Tema 4: Hooks

3.Hook useEffect

El hook **useEffect** en React es una herramienta poderosa y versátil que permite a los componentes funcionales manejar efectos secundarios. Los efectos secundarios son operaciones que pueden afectar otros componentes o que no se relacionan directamente con la salida del renderizado, como solicitudes de datos, suscripciones, temporizadores, manipulaciones del DOM y más.

3.1. Características Principales de useEffect

1. **Ejecución Después de Renderizar**: **useEffect** se ejecuta después de que el DOM se haya actualizado. Esto significa que no bloquea la renderización visual del componente.
2.
3. **Dependencias**: **useEffect** puede aceptar un segundo argumento: un array de dependencias. Este array determina cuándo se debe volver a ejecutar el efecto.
    - **Sin dependencias (useEffect(() => {...}))**: El efecto se ejecuta después de cada renderizado.
    - **Con un array vacío (useEffect(() => {...}, []))**: El efecto se ejecuta solo una vez, similar a **componentDidMount** en componentes de clase.
    - **Con dependencias (useEffect(() => {...}, [deps]))**: El efecto se ejecuta cada vez que cambia alguna de las dependencias en el array.
    -
4. **Limpieza**: **useEffect** puede retornar una función de limpieza que se ejecuta antes de desmontar el componente o antes de que el efecto se vuelva a ejecutar. Es útil para tareas de limpieza como cancelar suscripciones o temporizadores.

3.2. Ejemplo Básico

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

function Example() {
 const [data, setData] = useState(null);

 useEffect(() => {
  const fetchData = async () => {
   const response = await fetch('some-api-url');
   const result = await response.json();
   setData(result);
  };

  fetchData();
 }, []); // Se ejecuta solo una vez

 return (
  <div>
   {data ? <div>{data.someField}</div> : <div>Loading...</div>}
  </div>
 );
}
```

En este ejemplo, **useEffect** se usa para hacer una solicitud de datos a una API después de que el componente se monte. Dado que las dependencias son un array vacío, la solicitud se realiza solo una vez.

Tema 4: Hooks

3.3. Detalles de useEffect

useEffect es parte de los Hooks en React y se utiliza para ejecutar efectos secundarios en componentes funcionales. Un "efecto secundario" es cualquier operación que afecta algo fuera del ámbito del valor de retorno del componente, tales como:

- Solicitudes de datos a una API.
- Suscripciones a streams de datos o eventos.
- Modificaciones directas del DOM.
- Temporizadores o intervalos.

3.4. Sintaxis y Funcionamiento

La sintaxis básica de **useEffect** es:

```
useEffect(() => {
  // Lógica del efecto secundario
  return () => {
    // Lógica de limpieza
  };
}, [dependencias]);
```

- La función dentro de **useEffect** se ejecuta después de cada renderizado del componente, dependiendo de las dependencias especificadas.
- Si se devuelve una función desde dentro de **useEffect**, esta actuará como una operación de limpieza que se ejecuta antes de que el componente se desmonte o antes de que el efecto se vuelva a ejecutar.

3.5. Ejemplos

1. Efecto sin Dependencias

```
useEffect(() => {
  console.log('Este efecto se ejecuta después de cada renderizado');
});
```

2. Efecto con un Array Vacío de Dependencias

Este efecto se ejecuta una vez, similar a **componentDidMount** en componentes de clase.

```
useEffect(() => {
  console.log('Este efecto se ejecuta una sola vez, al montar el componente');
}, []);
```

3. **Efecto con Dependencias**

Se ejecuta cada vez que cambia el valor de **dependency**.

```
const [dependency, setDependency] = useState(0);

useEffect(() => {
  console.log('Este efecto se ejecuta cuando cambia `dependency`');
}, [dependency]);
```

4. **Efecto con Limpieza**

Útil para desuscripciones, borrado de temporizadores, etc.

```
useEffect(() => {
  const timer = setTimeout(() => {
    console.log('Efecto con temporizador');
  }, 1000);

  return () => clearTimeout(timer);
}, []);
```

3.6. Ejemplo Práctico: Fetching de Datos

```
function DataFetcher() {
  const [data, setData] = useState(null);

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

  if (!data) return <div>Cargando...</div>;

  return <div>{JSON.stringify(data)}</div>;
}
```

En este ejemplo, **useEffect** se usa para realizar una solicitud de datos cuando el componente se monta. Gracias al array vacío de dependencias, la solicitud se realiza solo una vez.

3.7. Conclusión

useEffect es un hook extremadamente útil y flexible para manejar todo tipo de efectos secundarios en tus componentes funcionales de React. Su capacidad de controlar cuándo se ejecutan los efectos secundarios y cómo se limpian los recursos es fundamental para un desarrollo efectivo y eficiente en React.

4.Hook useContext

El hook useContext en React es una herramienta que permite a los componentes funcionales acceder al contexto de React, una forma de pasar datos a través del árbol de componentes sin necesidad de pasar props manualmente en cada nivel. Esto es especialmente útil para compartir datos como preferencias del usuario, temas, información de autenticación, etc., a múltiples componentes en diferentes niveles de la jerarquía de componentes.

4.1 Cómo Funciona useContext

1.Crear un Contexto: Primero, necesitas crear un contexto utilizando React.createContext(). Esto devuelve un objeto Context que contiene dos componentes: Provider y Consumer.

```
const MyContext = React.createContext(defaultValue);
```

2.Proporcionar un Contexto: Usa el componente Provider del contexto creado para envolver los componentes que necesitan acceso a los datos del contexto. El Provider recibe una prop value, que es donde proporcionas los datos que quieres compartir.

```
<MyContext.Provider value={/* algún valor */}>
```

3.Consumir un Contexto: Para consumir el contexto en un componente funcional, utiliza el hook useContext y pasa el objeto de contexto como argumento. Esto te permite acceder a los datos del contexto directamente.

```
const value = useContext(MyContext);
```

4.2. Ejemplo Básico

```
import React, { useContext, createContext } from 'react';

// Crear un contexto
const UserContext = createContext();

function App() {
 // Proporcionar un contexto
 return (
  <UserContext.Provider value="Alice">
   <UserProfile />
  </UserContext.Provider>
 );
}

function UserProfile() {
 // Consumir un contexto
 const userName = useContext(UserContext);

 return <h1>User: {userName}</h1>;
}
```

En este ejemplo:

- Creamos un UserContext.
- En el componente App, usamos UserContext.Provider para pasar el valor "Alice" a los componentes anidados.
- En UserProfile, usamos useContext(UserContext) para acceder al valor del contexto, que es el nombre del usuario, y lo mostramos.

4.3. Ventajas de useContext

- Menos Prop Drilling: Reduce la necesidad de pasar props a través de varios niveles de componentes.
- Código Más Limpio y Mantenible: Facilita la gestión y el acceso a datos compartidos, lo que conduce a un código más limpio y mantenible.
- Rendimiento Optimizado: React realiza optimizaciones en el contexto para evitar renderizados innecesarios.

4.4. Conclusión

useContext es un hook valioso en React para acceder y manejar datos en un contexto compartido, lo que simplifica significativamente el flujo de datos en aplicaciones complejas y mejora la organización y mantenibilidad del código.

5.Hook useRef

El hook useRef en React es una herramienta que permite a los componentes funcionales mantener una referencia mutable a un objeto a lo largo del ciclo de vida del componente. useRef es útil para varias situaciones, como acceder a un elemento del DOM, almacenar un valor mutable que no causa una re-renderización cuando cambia, y mantener una referencia consistente a lo largo de re-renderizaciones.

5.1. Cómo Funciona useRef

Creación de la Referencia: Se crea una referencia utilizando useRef() y se le puede asignar un valor inicial.

```
const myRef = useRef(initialValue);
```

Esto devuelve un objeto mutable con la propiedad .current que se inicializa con el initialValue proporcionado.

Acceso y Modificación: Puedes acceder y modificar myRef.current en cualquier momento, y el valor se mantendrá consistente a través de los renderizados del componente.

Uso con Elementos del DOM: useRef es comúnmente usado para obtener una referencia a un elemento del DOM. Puedes pasar myRef al atributo ref de un elemento JSX para acceder directamente a ese nodo del DOM.

5.2. Ejemplo de Uso con Elementos del DOM

```
import React, { useRef } from 'react';

function TextInputWithFocusButton() {
 const inputEl = useRef(null);

 const onButtonClick = () => {
  // `current` apunta al nodo de entrada de texto montado
  inputEl.current.focus();
 };

 return (
  <>
   <input ref={inputEl} type="text" />
   <button onClick={onButtonClick}>Focus the input</button>
  </>
 );
}
```

En este ejemplo, useRef se utiliza para mantener una referencia al elemento de entrada de texto. Cuando se hace clic en el botón, se accede a inputEl.current para enfocar programáticamente el input.

5.3. Uso para Almacenar un Valor Mutable

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

function Timer() {
 const [timer, setTimer] = useState(0);
 const intervalRef = useRef();

 useEffect(() => {
  intervalRef.current = setInterval(() => {
   setTimer((t) => t + 1);
  }, 1000);

  return () => clearInterval(intervalRef.current);
 }, []);

 return (
  <div>
   Timer: {timer}
   <button onClick={() => clearInterval(intervalRef.current)}>
    Stop Timer
   </button>
  </div>
 );
}
```

En este ejemplo, useRef se usa para mantener una referencia al intervalo del temporizador. Esto permite que el intervalo sea detenido correctamente cuando se desmonta el componente o cuando se hace clic en el botón.

5.4.Conclusiones

useRef es versátil y se utiliza para acceder a elementos del DOM y para almacenar valores mutables que no deben causar re-renderizaciones.

Proporciona un objeto mutable .current que se mantiene consistente a lo largo de los renderizados.
Es una herramienta importante en el arsenal de hooks de React para manejar ciertos tipos de estado y referencias de manera eficiente.

6.Hook useReducer

El hook useReducer en React es una alternativa al hook useState y es particularmente útil cuando tienes una lógica de estado compleja o cuando el próximo estado depende del estado anterior. useReducer también facilita la centralización de la lógica de manejo del estado, lo que puede hacer tu código más legible y fácil de mantener, especialmente en componentes grandes o complejos.

6.1. Cómo Funciona useReducer

1. **Definición del Reductor**: Un "reductor" es una función que toma el estado actual y una acción, y devuelve un nuevo estado. Tiene la forma (state, action) => newState.

```
const reducer = (state, action) => {
  switch (action.type) {
    case 'accion1':
      return { ...state, /* nuevos cambios en el estado */ };
    // otros casos
    default:
      throw new Error();
  }
};
```

2. **Inicialización de useReducer**: useReducer se inicializa con el reductor y un estado inicial.

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

Aquí, state es el estado actual, y dispatch es una función que puedes llamar para enviar una acción al reductor.

3. **Envío de Acciones**: Las acciones son objetos que describen qué debe cambiar. Usualmente tienen un campo type que indica el tipo de acción.

```
dispatch({ type: 'accion1', /* otros datos */ });
```

6.2. Ejemplo Básico

```
import React, { useReducer } from 'react';

const initialState = { count: 0 };

function reducer(state, action) {
  switch (action.type) {
    case 'increment':
      return { count: state.count + 1 };
    case 'decrement':
      return { count: state.count - 1 };
    default:
      throw new Error();
  }
}

function Counter() {
  const [state, dispatch] = useReducer(reducer, initialState);

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

En este ejemplo, useReducer se utiliza para manejar el estado de un contador. La función reducer maneja dos acciones: increment y decrement. Las acciones son enviadas usando dispatch desde los botones en el componente.

6.3. Ventajas de useReducer

- **Manejo de Estados Complejos**: Es más adecuado para gestionar estados complejos y anidados.
- **Mantenibilidad y Legibilidad**: Centraliza la lógica de manejo del estado y puede hacer que el componente sea más legible.
- **Optimización del Rendimiento**: Puede ser más eficiente que useState para ciertos patrones de actualización, especialmente para actualizaciones frecuentes o complejas.

6.4.Conclusión

useReducer es un hook poderoso para el manejo de estado en componentes funcionales de React, particularmente útil para lógicas de estado más complejas y para mejorar la organización del código. Su similitud con el patrón Redux también lo hace atractivo para quienes están familiarizados con esa biblioteca.

Tema 4: Hooks

7.Hook useCallback

El hook useCallback en React es una herramienta diseñada para optimizar el rendimiento de tu aplicación, especialmente útil en componentes que renderizan una gran cantidad de elementos o que utilizan componentes hijos pesados. useCallback devuelve una versión memorizada de una función callback, lo que ayuda a evitar renderizados innecesarios.

7.1. Cómo Funciona useCallback

Definición de useCallback:

useCallback toma una función inline y un array de dependencias.

React memorizará la función pasada y solo cambiará si una de las dependencias ha cambiado.

```
const memoizedCallback = useCallback(
  () => {
    // Hacer algo con las dependencias
  },
  [dependencias], // Solo se vuelve a calcular si las dependencias cambian
);
```

Uso de la Función Memorizada:

Puedes usar memoizedCallback en tu componente o pasarlo a componentes hijos.
La función no se volverá a crear a menos que una de las dependencias cambie.

7.2. Ejemplo de Uso

Supongamos que tienes un componente que recibe una función como prop y que esta función depende del estado del componente padre:

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

function MyComponent({ onAction }) {
  // Renderiza algo que usa onAction
}

function ParentComponent() {
  const [count, setCount] = useState(0);

  const handleAction = useCallback(() => {
    // Hacer algo con 'count'
    console.log(count);
  }, [count]); // handleAction solo se vuelve a crear si 'count' cambia

  return (
    <>
      <MyComponent onAction={handleAction} />
      <button onClick={() => setCount(c => c + 1)}>Incrementar</button>
    </>
  );
}
```

En este ejemplo, useCallback asegura que handleAction no se vuelva a crear cada vez que ParentComponent se renderiza, a menos que el estado count haya cambiado. Esto puede ser útil para evitar renderizados innecesarios de MyComponent si depende de la igualdad de referencia de sus props para evitar actualizaciones costosas.

7.3. Ventajas de useCallback

- Optimización del Rendimiento: Reduce el número de renderizados innecesarios en componentes hijos, al evitar la creación de nuevas instancias de funciones en cada renderizado.
- Memorización de Funciones: Útil cuando pasas callbacks a componentes optimizados con React.memo o cuando la igualdad de referencia es importante.

7.4. Conclusión

useCallback es una herramienta de optimización. No siempre es necesario, pero puede ser muy útil en situaciones donde el rendimiento es una preocupación, especialmente en componentes complejos o pesados. Sin embargo, es importante usarlo sabiamente, ya que agregar demasiadas memorizaciones puede llevar a una sobrecarga de rendimiento debido a la complejidad adicional.

8.Hook useMemo

El hook useMemo en React es una herramienta para optimizar el rendimiento de los componentes, similar a useCallback. Sin embargo, mientras que useCallback memoriza una función callback, useMemo se utiliza para memorizar un valor calculado, es decir, el resultado de una función.

8.1. Cómo Funciona useMemo

Definición de useMemo:

useMemo toma una función que calcula un valor y un array de dependencias.
React ejecutará esta función y memorizará su resultado. La próxima vez que el componente se renderice, si las dependencias no han cambiado, React devolverá el valor memorizado en lugar de recalcularlo.

```
const memoizedValue = useMemo(() => {
  // Cálculo o lógica pesada
  return computeExpensiveValue(a, b);
}, [a, b]); // Solo recalcula si 'a' o 'b' cambian
```

Uso del Valor Memorizado:

Puedes usar memoizedValue en tu componente como cualquier otro valor.
El valor solo se vuelve a calcular si las dependencias especificadas cambian.

8.2. Ejemplo de Uso

Supongamos que tienes una función costosa que quieres ejecutar solo cuando ciertas dependencias cambien:

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

function ExpensiveComponent({ propA, propB }) {
  const expensiveValue = useMemo(() => {
    // Una operación costosa
    return computeExpensiveValue(propA, propB);
  }, [propA, propB]);

  return <div>{expensiveValue}</div>;
}

function computeExpensiveValue(a, b) {
  // Alguna lógica compleja que tarda en computar
  console.log('Computando valor costoso');
  return a + b; // Ejemplo simplificado
}
```

En este ejemplo, computeExpensiveValue solo se llama cuando propA o propB cambian, evitando cálculos innecesarios en otros renderizados.

8.3 Ventajas de useMemo

- Optimización del Rendimiento: Es útil para evitar cálculos costosos en cada renderizado.
- Prevención de Renderizados Innecesarios: Puede ayudar a evitar renderizados innecesarios de componentes hijos que dependen del valor calculado.

8.4. Conclusión

useMemo es una herramienta útil para mejorar el rendimiento en ciertas situaciones, especialmente cuando se trabaja con cálculos o procesos costosos que no necesitan ejecutarse en cada renderizado. Sin embargo, su uso debe ser justificado, ya que el sobrecosto de memorización puede superar los beneficios en casos donde los cálculos no son realmente costosos o no cambian frecuentemente. Su uso correcto y mesurado puede ser muy beneficioso para optimizar componentes en React.