Consumo de APIs con el hook UseEffect y referenciar con useRef

Objetivos

- Conocer el hook useEffect
- Ciclo de vida de un componente
- Aprender a consumir APIs con el hook useEffect
- Conocer el hook useRef

¿Qué es useEffect?

useEffect es un hook en React, que se utiliza para especificar una función que React ejecutará después de que se haya realizado una renderización y se haya actualizado el DOM.

El hook useEffect toma dos argumentos: una función que contiene el código que se ejecutará y un arreglo de dependencias opcional. El arreglo de dependencias se utiliza para especificar qué variables deben cambiar para que el efecto se vuelva a ejecutar. Si no se proporciona un arreglo de dependencias, el efecto se ejecutará después de cada renderización.

useEffect sin dependencias

En este ejemplo, se utiliza useEffect sin un arreglo de dependencias. Esto significa que el efecto se ejecutará después de cada renderización del componente, sin importar si alguna variable específica ha cambiado o no. Esto es útil cuando deseas realizar alguna acción después de cada renderización, independientemente de las variables que se utilicen en el componente.

```jsx
import { useState, useEffect } from 'react';  4.2k (gzipped: 1.8k)

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

  useEffect(() => {
    // Este código se ejecutará después de cada renderización
    console.log('Efecto sin dependencias ejecutado');
  });

  return (
    <div>
      <p>Contador: {count}</p>
      <button onClick={() => setCount(count + 1)}>Incrementar</button>
    </div>
  );
}

export default ExampleComponentWithoutDependencies;
```

useEffect con dependencias

En este ejemplo, utilizamos useEffect con un arreglo de dependencias. Esto significa que el efecto solo se ejecutará cuando alguna de las variables especificadas en el arreglo de dependencias haya cambiado. Esto es útil cuando deseas realizar acciones específicas solo cuando ciertas variables cambian.

```jsx
import { useState, useEffect } from 'react';  4.2k (gzipped: 1.8k)

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

  useEffect(() => {
    // Este código se ejecutará solo cuando 'count' cambie
    console.log('Efecto con dependencias ejecutado');
  }, [count]);

  return (
    <div>
      <p>Contador: {count}</p>
      <button onClick={() => setCount(count + 1)}>Incrementar</button>
    </div>
  );
}

export default ExampleComponentWithDependencies;
```

En este caso, el efecto solo se ejecutará cuando count cambie. Esto es útil cuando tienes lógica que debe ejecutarse en respuesta a cambios específicos en el estado del componente.

El arreglo de dependencias es una herramienta importante para controlar cuándo se ejecuta un efecto en React, y es importante usarlo de manera adecuada para evitar ejecuciones innecesarias y optimizar el rendimiento de tus componentes.

Relación entre useEffect y el ciclo de vida de un componente

Ciclo de vida de un componente

El ciclo de vida de un componente en React se refiere a las diferentes etapas o momentos que atraviesa un componente desde su creación hasta su eliminación. Cada una de estas etapas proporciona oportunidades para realizar acciones específicas, como la configuración inicial, la actualización en respuesta a cambios de estado o props, y la limpieza antes de que el componente se elimine.

Etapas principales en el ciclo de vida de un componente

Montaje (Mounting): Esta es la primera escena cuando se crea el componente y se muestra en la pantalla por primera vez. Aquí, puedes hacer cosas como preparar datos iniciales y configurar cosas.

Ejecutando useEffect en solo en el montaje

Si deseas ejecutar un useEffect solo en el montaje de un componente y no en las actualizaciones posteriores, puedes lograrlo proporcionando un arreglo de dependencias vacío ([]) como segundo argumento en useEffect. De esta manera, el efecto se ejecutará solo una vez, después de que el componente se haya montado en el DOM.

```
import { useEffect } from 'react';  4.1k (gzipped: 1.8k)

function ExampleComponent() {
  useEffect(() => {
    // Este código se ejecutará solo después del montaje del componente
    console.log('Efecto de montaje ejecutado');
  }, []); // El arreglo de dependencias está vacío

  return <div>Contenido del componente</div>;
}

export default ExampleComponent;
```

Actualización (Updating): Esto es como las escenas que ocurren cuando el componente ya está en pantalla y algo cambia, como recibir nuevas instrucciones o información. Puedes decidir si el componente debe actualizarse o no.

Desmontaje (Unmounting): Esta es la última escena cuando el componente se quita de la pantalla. Aquí, puedes limpiar cualquier cosa que no necesites más.

Manejo de errores (Error Handling): Si algo sale mal mientras el componente está en pantalla, esta escena maneja la situación y puede mostrar un mensaje de error en lugar de mostrar el componente roto.

Ahora que conocemos las principales etapas de un componente debemos saber que useEffect puede actuar en las 3 primeras etapas.

Actualizaciones problemáticas

Las ejecuciones de useEffect después de cada actualización pueden ser problemáticas en las siguientes situaciones:

Consumo excesivo de recursos: Si tienes un useEffect sin un arreglo de dependencias ([]) y no controlas adecuadamente su ejecución, podría ejecutarse en cada renderizado del componente. Esto podría llevar a una cantidad excesiva de llamadas a la función de efecto y, en última instancia, al consumo excesivo de recursos.

Por ejemplo, si haces una solicitud de red en cada renderizado sin ninguna restricción, podrías generar una gran cantidad de solicitudes innecesarias.

Efectos secundarios innecesarios: Si no controlas las condiciones bajo las cuales se ejecuta useEffect, podrías realizar efectos secundarios (como solicitudes de red o actualizaciones del DOM) cuando no sea necesario. Esto puede afectar negativamente el rendimiento y la experiencia del usuario.

Consumiendo APIs con useEffect

A continuación, veremos paso a paso como consumir una API con react y el hook useEffect

Paso 1:

Importar useEffect y useState: Asegúrate de importar useEffect y useState desde React.

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

Paso 2:

Utilizar useEffect para realizar la solicitud a la API. Dentro de useEffect, puedes realizar la solicitud a la API utilizando la función fetch u otra biblioteca como Axios. Asegúrate de que esta solicitud se realice de manera asincrónica y actualice el estado con los datos recibidos.

```jsx
import { useState, useEffect } from "react";  4.2k (gzipped: 1.8k)

const CallApi = () => {
  const [data, setData] = useState([]);
  useEffect(() => {
    // Definir la URL de la API que deseas consumir
    const apiUrl = "https://ejemplo.com/api";

    fetch(apiUrl)
      .then((response) => response.json())
      .then((data) => {
        // Actualizar el estado con los datos de la API
        setData(data);
      })
      .catch((error) => {
        // Manejar errores de la solicitud
        console.error("Error al obtener datos de la API:", error);
      });
  }, []);

  return (
    <div>{data}</div>
  )
}

export default CallApi
```

Paso 3:

Manejar el ciclo de vida de la solicitud:
Dentro de useEffect, es importante asegurarse de que la solicitud se realice solo una vez en el montaje del componente (cuando el arreglo de dependencias esté vacío []). Si deseas que la solicitud se realice nuevamente en respuesta a cambios específicos (por ejemplo, cuando un prop cambie), debes agregar esas variables al arreglo de dependencias.

```jsx
useEffect(() => {
  // Realizar la solicitud a la API aquí
}, [data]);
```

Paso 4:

Renderizar los datos en tu componente:
Finalmente, puedes renderizar los datos en tu componente. Ten en cuenta que data es el estado que contiene los datos de la API, así que puedes mapearlos y mostrarlos en tu JSX.

```jsx
return (
  <div>
    <h1>Datos de la API</h1>
    <ul>
      {data.map((item) => (
        <li key={item.id}>{item.name}</li>
      ))}
    </ul>
  </div>
);
```

Ahora que ya sabemos cómo consumir una API con useEffect vamos a hacer solicitudes a una API real para ello utilizaremos la API de Super Hero.

El hook useRef

useRef es un hook de React que se utiliza para acceder al DOM (Document Object Model) o para mantener referencias a elementos HTML u otros valores que no deben causar una re-renderización cuando cambian.

Principalmente, useRef se utiliza para los siguientes propósitos:

Acceso al DOM: Puedes usar useRef para obtener una referencia a un elemento del DOM y luego interactuar directamente con él. Por ejemplo, puedes cambiar su contenido o aplicar estilos sin necesidad de cambiar el estado de React o provocar una re-renderización.

```jsx
import {useRef, useEffect} from "react";  4.2k (gzipped: 1.8k)

const RefDom = () => {
  const myRef = useRef();

  useEffect(() => {
    myRef.current.innerHTML = "Este es un elemento modificado";
  }, []);

  return <div ref={myRef}>Este es un elemento de referencia</div>;
};

export default RefDom;
```

Mantener valores sin causar re-renderización: useRef también se utiliza para mantener valores que no deben causar una re-renderización cuando cambian. A diferencia de los estados (useState), los cambios en un objeto ref no provocarán una re-renderización. Esto es útil para mantener valores que no están destinados a afectar la interfaz de usuario.

```jsx
import { useRef, useEffect } from "react";  4.2k (gzipped: 1.8k)

const RefValues = () => {
  // el contador parte en 0 y se incrementa en 1 cada vez que se renderiza el componente
  const count = useRef(0);
  //el contador no cambia en useEffect al cargar el componente por primera vez
  // pero si cambia en useEffect cuando se actualiza el componente
  useEffect(() => {
    count.current = count.current + 1;
    console.log(`El valor de count es: ${count.current}`);  ×3 'El valor de count es: 2'
  }, []);

  return <div>Contador: {count.current}</div>;
};

export default RefValues;
```

Si probaste el código te daras cuenta de que el contador no cambia al cargar la página porque el efecto se ejecuta después de que el componente se monta, y en ese momento, el valor de count.current ya está configurado como 0 debido a la inicialización que se hizo con useRef.

En resumen, useRef es una herramienta que se utiliza en casos específicos donde necesitas interactuar con el DOM, mantener valores persistentes o evitar renderizaciones innecesarias. Su principal utilidad es mejorar el rendimiento y el acceso a elementos del DOM sin causar una actualización innecesaria en la interfaz de usuario.

Felicitaciones llegaste al final del primer módulo de React, a continuación se va a entregar la prueba que corresponde a esta unidad te deseo mucha suerte.