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.

Antes, para utilizar componentes con estados era necesario crearlos desde un **class**. Con los hooks es posible usar estados en componentes creados desde una function

**CREANDO EL PROYECTO**

Para esta sección, se copiaran algunas carpetas del proyecto anterior y se realizará una especie de “restaurado” para que provea al proyecto de los archivos necesarios para utilizar react.

Los archivos/carpetas a copiar son: public, src, package, package.lock, .gitignore

Una vez abierta la carpeta en vscode, es necesario abrir un terminal y ejecutar **npm install**

*found 3 moderate severity vulnerabilities*

*run `npm audit fix` to fix them, or `npm audit` for details*

este es un mensaje que puede aparecer

**USE STATE**

Este hook, se importa de la forma:

import { useState } from 'react'

si se imprime en consola, se obtiene un array, en el cual la primera posición es el valor del estado, y la segunda una funcion hecha para modificar el valor de este estado.

Para desestructurarlo y usarlo puede usarse:

const [state, setState] = useState(0);

esta línea además inicializará el estado en 0

Es importante que el estado se modifique cuando sea necesario. Si se renderiza por renderizar, consumirá ram y hará fallar la aplicación.

**USE EFFECT**

El effect controla “el ciclo de vida” de la aplicación. UseEffect se ejecuta cada vez que se renderiza el component.

Use effect es una funcion y recibe, como primer parámetro una funcion de callback y como segundo una lista de dependencias.

Dentro de la funcion se incluye lo que se ejecutará y en las dependencias, las “condiciones” para que esto se haga.

Esto debido a que cada vez que se renderiza, se realizan peticiones HTTP. Con esto lo que se intenta es reducir las peticiones al mínimo.

**FORMULARIO**

Para explicar el funcionamiento del hook **useEffect** se aplicará a un ejemplo de formulario.

En este proyecto se está utilizando Bootstrap, y una de las mayores complicaciones es el no cierre de etiquetas self-closing y el uso del atributo class en lugar de className en los elementos.

Para esto, puede copiarse el código de Bootstrap y pegarlo en un convertidor de HTML a JSX

<https://magic.reactjs.net/htmltojsx.htm>

Para el ejemplo solo se conservará un <form> con solo 1 <input> y un <button> de type submit.

La primer acción es asignarle una funcion al evento **onSubmit** del <form> y prevenir la Accion por defecto para evitar recargar el sitio.

**FETCH Y ASYNC**

Una forma de realizar la peticion en el **userEffect** podría ser declarando una función por fuera e invocándola dentro de esta (llamando a getUsers).

  const getUsers = async () => {

    const res = await fetch("https://jsonplaceholder.typicode.com/users");

    const data = await res.json();

    setState(data);

  };

  useEffect(() => {

    getUsers();

  }, []);

El problema de esto es que se recomienda que la funcion dentro de **useEffect** no sea async, como lo es en este caso. Por tanto, será necesario ejecutar la petición con las promesas y el .then()

**USE EFFECT**

Este es el código del ejemplo:

Consta de 2 estados, uno para guardar los datos traídos de la petición, y uno para usar como dependencia para “oir” los cambios.

Además, hay 2 useEffects, uno que se ejecutará únicamente al inicio y otro, que además de ejecutarse inicialmente, comenzará a ejecutarse cada vez que la dependencia cambie.

El resto del código necesario son valores para los atributos del elemento <input> con el que se estará jugando.

  const [state, setState] = useState([]); //estado para guardar users

  const [id, setId] = useState(0); //dependencia a oir para cambiar

  //este use Effect se ejecuta al inicio

  useEffect(() => {

    fetch("https://jsonplaceholder.typicode.com/users")

      .then((res) => res.json())

      .then((data) => setState(data));

  }, []);

  //este tambien y espera por cambios en id para volver a ejecutarse

  useEffect(() => {

    setId(id);

  }, [id]);

          <input

            value={id}

            onChange={(e) => setId(e.target.value)}

            type="text"

            className="form-control"

            id="exampleInputEmail1"

            aria-describedby="emailHelp"

          />

Como value (o sea lo que hay en el input) se establece el estado actual de **id**

Y como envento **onChange** se “setea” el valor que se tome del input. Desde el evento se envía el value al estado, y de el estado se establece el atributo value.  
**value 🡪 onChange 🡪 state 🡪 value**

0->1 🡪 setea 1 🡪 estado 1 🡪 value 1

**LA FORMA CORRECTA DE USAR USE EFFECT**

La forma correcta sería, utilizando una dependencia (“escuchándola”) y realizando acciones dentro del effect.

En este caso se modificará el estado del componente en base al valor del <input> y se escuchara por el effect en base al estado.

          <input

            value={state}

            onChange={(e) => setState(e.target.value)}

            type="text"

            className="form-control"

            id="exampleInputEmail1"

            aria-describedby="emailHelp"

          />

  useEffect(() => {

    if (state === "123") {

      window.addEventListener("click", evento);

    }

    return () => {

      window.removeEventListener("click", evento);

    };

  }, [state]);

Como se puede ver, en el use effect, se corrobora que el estado cumpla con cierta condición para añadir el escuchador al click, pero además, en caso de que no sea así, se remueve el escuchador

Esta es la declaración del **evento** para no repetirle en cada funcion. Además puede no funcionar si no se hace de esa manera.

  const evento = () => {

    console.log("clicking");

  };

**LAYOUTEFFECT**

useLayoutEffect

La firma es idéntica a useEffect, pero se dispara de forma síncrona después de todas las mutaciones de DOM. Use esto para leer el diseño del DOM y volver a renderizar de forma sincrónica. Las actualizaciones programadas dentro de useLayoutEffect se vaciarán sincrónicamente, antes de que el navegador tenga la oportunidad de pintar.

Este es un ejemplo de como usarle:

  //Estado 1

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

  //estado 2

  const [lenght, setLenght] = useState(0);

  //data

  const newData = [{ nombre: "agustin", apellido: "stringa" }];

  //a los 5 seg de cargar el sitio actualiza este estado

  useEffect(() => {

    setTimeout(() => {

      setState(newData);

    }, 5000);

  }, []);

  //al cambiar el otro, este se modifica y toma como valor su "largo"

  useLayoutEffect(() => {

    setLenght(state.length);

  }, [state]);

Este ejemplo lleva 2 estados. El primero, **state** que guarda un array. El segundo, que es el largo del array del primero.

Lo que se hace, con el useEffect, es, luego de 5 segundos, modificar este estado y cargar un array creado llamado **newData**

El **useLayoutEffect** lleva como dependencia el estado, por tanto, al cambiar **state** se lanza el **useLayouteffect** ya que el largo de state, cambió.

**USE REF**

Almacena una referencia a un elemento. EN otras palabras, simplifica el guardar en una variable el **document.getElementById()**

Se inicializa de la forma

const ref = useRef(null); // por parámetro se indica el valor inicial para la ref

y requiere la importación:

import { useRef } from "react";

Dentro de los elementos del return de un componente, puedo incluir el atributo **ref** (propio de react) para asignarle una referencia.

Este objeto “ref” contiene una propiedad llamada “current” que es el valor actual para la misma.

Es importante saber que la referencia se declara como tal y se agrega a un atributo de un elemento, pero su valor en **ref.current** puede variar para que se opere en base a este.

Ejemplo:

import React from "react";

import { useRef } from "react";

const Ref = () => {

  const ref = useRef(null);

  const handleRef = () => {

    // const area = document.querySelector("#area").value;

    ref.current.value = "hi world";

    console.log(ref);

  };

  return (

    <>

      <h1 className="bg-info p-3" onClick={handleRef}>

        Use ref

      </h1>

      <textarea ref={ref} id="area" placeholder="escribe aqui..."></textarea>

    </>

  );

};

export default Ref;

como se puede ver en el evento asociado al click del <h1> se accede a **ref.current**, que en ese momento es el textarea#area y se le modifica su propiedad **.value**

Si en otro bloque de código modificásemos **ref.current**, pasaríamos a tener en esta propiedad una referencia a otro elemento y podríamos modificarle.

La principal utilidad de **useRef** es acceder a elementos del DOM imperativamente, sin forzar un renderizado. Por ejemplo, para dar focus a un input mediante un botón.

11-10-21

Si se quisesen hacer notar los cambios en el hook Ref, sería necesario renderizar el componente. Una forma de verlo es declarar un estado y modificarle para que se renderice el component y notar que la Ref ahora a apunta a otro elemento del DOM.

**memo**

memorize se utiliza para “guardar” en memoria una “versión del componente” y no renderizarle innecesariamente, por ejemplo, cuando cambia un estado del componente padre que no le involucre.

Para este ejemplo se crea un componente Memo que contiene el componente Dato. Estos se relacionan por medio del useState llamado **counter** y esta es la estructura.

import React, { useState } from "react";

import Dato from "./Dato";

const Memo = () => {

*//estado contador*

  const [counter, setCounter] = useState(0);

  return (

    <>

      <p>

        Counter: <Dato *value*={counter}></Dato>

      </p>

      <button

*onClick*={() => {

          setCounter(counter + 1);

        }}

      >

        Aumentar counter

      </button>

const Dato = React.memo((*props*) => {

  console.log("me renderice");

  return <span>{*props*.value}</span>;

});

El componente dato recibe se invoca con el atributo value=”” el **counter** que es uno de los estados del componente padre.

En este caso se recibe el objeto props, aunque tranquilamente podría destructurarse y presentarse de esta manera:

const Dato = React.memo(({ *value* }) => {

  console.log("me renderice");

  return <span>{*value*}</span>;

});

Se envía un mensaje a consola que dice “me reenderice” para hacer notar que, cuando se incluye otro estado en el componente padre y se modifica, el componente “hijo” se vuelve a renderizar a pesar de no tener modificaciones.

(código en el componente Memo)

*//estado independiente*

  const [fondo, setFondo] = useState(true);

  return (

    <>

      <p>

        Counter: <Dato *value*={counter}></Dato>

      </p>

      <button

*onClick*={() => {

          setCounter(counter + 1);

        }}

      >

        Aumentar counter

      </button>

      <br />

      <button

*onClick*={() => {

          handleFondo();

        }}

      >

        State independiente

      </button>

    </>

  );

Ahora se agrega un nuevo botón que se encarga de alternar el estado del useState **fondo.** Este cambio de estado obliga a renderizar todo el componente y por tanto el componente hijo “memo”.

La solución es declarar el component Dato de la siguiente manera:

import React, { memo } from "react";

const Dato = memo(({ *value* }) => {

  console.log("me renderice");

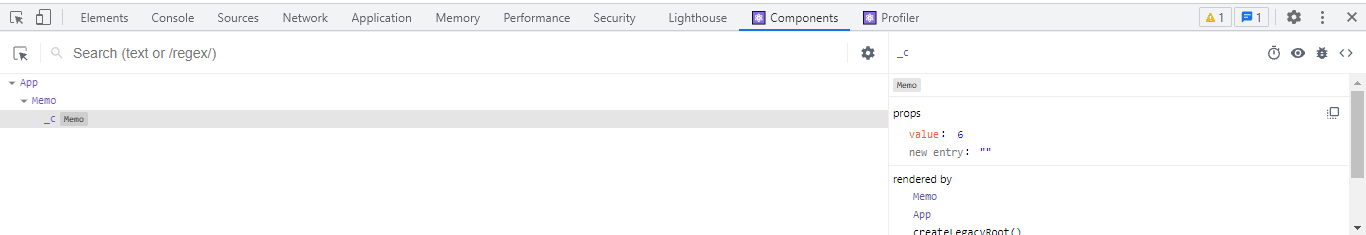
  return <span>{*value*}</span>;

});

export default Dato;

al importar memo, este recibe “el componente” como parámetro, por tanto Dato queda definido como se ve en el código anterior.

Ahora esta es la visión que se obtiene desde la pestaña de components en React Dev Tools:



Y al modificar el “estado independiente”, **fondo**, no ocurren renderizaciones en el componente memo.

**useMemo**

Lo mismo que con el “resultado” de un componente, puede ocurrir con el resultado de una funcion, para esto, vamos a utilizar useMemo.

En este caso se realiza la importación en el component padre:

import React, { useState, useMemo } from "react";

la funcion que se va a utilizar es la siguiente:

  const procesar = (*limite*) => {

    for (let i = 0; i < *limite*; i++) {

      console.log(`numero: ${i}`);

    }

Recibe un limite e itera desde 0 hasta ese limite. Esto para mostrar que useMemo puede utilizarse para evitar acciones que realenticen la app obsoletamente.

Si dentro del componente reendirazasemos:

{procesar(counter)}

Cada vez que modifiquemos el state independiente, se ejecutaría esta funcion, aunque el **counter** no haya cambiado. Para solucionar esto, se hace:

  const memorizar = useMemo(() => procesar(counter), [counter]);

se guarda en una constante memorizar el resultado del useMemo, este hook funciona muy similar a useEffect, recibe una funcion y un array de dependencias. Ahora, deberíamos renderizar en el return lo sig:

{memorizar}

Entonces, esto se ejecutará solo cuando cambie el **counter** y podremos alternar estados ajenos sin la necesidad de consumir memoria en operaciones innecesarias.

**UseCallback**

Otro inconveniente de optimización es el que ocurre cada vez que se renderiza un componente, por ejemplo, con la declaración de funciones, que vuelven a “crearse” aunque sean iguales a las que se han utilizado anteriormente.

useCallback recibe una funcion que se guardará en memoria y recibe como dependencia una funcion. En este ejemplo, se crea un componente llamado Callback.jsx y se utiliza useCallback en el componente padre. El componente Callback.jsx será el que ahora se encargará de aumentar el “contador” en 1.

const Callback = memo(({ *funcion* }) => {

  return (

    <>

      {console.log("Callback.jsx: agregando1")}

      <span>Usando useCallback para almacenar funciones</span>

      <button *onClick*={*funcion*}>Aumentar counter</button>

    </>

  );

});

export default memo(Callback);

el componente recibe una funcion como props, que es la que ejecutará al hacerse click sobre el mismo.

Desde el componente padre, se implementa de esta forma:

Esta es la funcion que se pasa como prop al componente:

  const add = useCallback(() => {

    setCounter((*actual\_value*) => *actual\_value* + 1);

  }, [setCounter]);

Y asi se invoca al componente <callback/>

<Callback *funcion*={add}></Callback>

Gracias al memo en el componente Callback, cuando se actualice el state **counter** la siguiente línea no se ejecutará:

{console.log("Callback.jsx: agregando1")}

Tampoco el componente se verá afectado por otros cambios en el componente padre, solo se renderizará o afectará si la dependencia lo indica.