Hooks

Los hooks son una colección de distintas funciones, cada una con un propósito muy claro y descriptivo y que están pensadas para ser utilizadas dentro de componentes funcionales, permitiéndonos no solo disponer de un manejo de state o control del ciclo de vida entre nuestros componentes funcionales si no que además, nos va a permitir reutilizar lógica entre nuestros componentes e incluso crear nuestros propios hooks.

Los hooks siguen una convención de nombre que emplea **use** seguido de un descriptor para el hook, por lo que todos los hooks son useHookName, como por ejemplo useState que es probablemente el hook más importante.

USESTATE

useState es el equivalente al state tradicional y nos va a permitir acceder a los valores del state, definir un state por defecto y también modificarlo, salvo que para ello no vamos a disponer del método setState tradicional ya que cada variable de state que definamos (count) tendra su propia función de modificación (setCount) como vemos en el siguiente ejemplo:

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

const UseStateComponent = () => {
  //se define la variable count y su setter setCount utilizando destructuracion de Arrays
  const [count, setCount] = useState(0);// 0 es el valor inicial del state
```

```
  return (
     <div>
       Count: { count }
       <button onClick={() => setCount(count + 1)}>+</button>
       <button onClick={() => setCount(count - 1)}>-</button>
       <button onClick={() => setCount(0)}>Reset</button>
     </div>);
}

export default UseStateComponent;
```

Probablemente en el primer vistazo, te haya parecido confusa la forma en la que se definen variables del state y esto se debe a que los hooks hacen un uso intensivo de la destructuración de arrays y si no estás acostumbrado a ella, al principio cuesta un poco. Podemos definir tantas propiedades como queramos (como veremos en otros ejemplos) llamando a useState para cada una de las variables o podemos definir un objeto, pero en líneas generales es mejor usar variables independientes.

USEEFFECT

useEffect es un hook que está pensado para ejecutar side effects y que en esencia son operaciones que suceden al margen del render. useEffect es el lugar adecuado para realizar llamadas a servicios, levantar listener de eventos, etc por lo que viene a ser el sustituto de componentDidMount y componentDidUpdate ya que useEffect se ejecuta **después** del propio render.

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

const UseEffectComponent = () => {
  //usamos useState para definir mouseEvent y su setter
  let [mouseEvent, setMouseEvent] = useState(0);

  useEffect(() => {
    //useEffect se ejecutará después del render y en él establecemos el listener
    document.addEventListener("mousemove", setMouseEvent, false);
  });

  return (
   <fieldset>
      <div>
        X: { mouseEvent.clientX }
        Y: { mouseEvent.clientX }
      </div>
    </fieldset>);
}
```

```
export default UseEffectComponent;
```

El funcionamiento de los effect puede llevar a confusión al principio ya que es importante tener en cuenta que se ejecutarán después de cada render, y si de la ejecución el effect se deriva un nuevo render, el effect volverá a ejecutarse de nuevo y así hasta el infinito. Para evitar estas situaciones los effect disponen de un mecanismo para asegurarse que solo se ejecuta una vez.

Vamos a explicarlo un poco mejor con un ejemplo de mundo real. Pensemos que tenemos un array vacío de datos y queremos llenarlo con una lista de elementos que viene de una llamada a un servicio, como por ejemplo una lista de posts, pero lógicamente solo queremos hacer esa llamada al servicio una única vez usando useEffect. Para ello, le diremos a useEffect que el valor inicial es un array vacío, que llenaremos con los posts, y useEffect comprobará si el valor ha cambiado para no volver a ejecutarse:

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

const UseEffectApiRequest = () => {
  //Definimos el array de posts
  let [posts, setPosts] = useState([]);
  //Queremos mostrar un loading mientras carga
  let [isLoading, setIsLoading] = useState(false);

  useEffect(() => {
    setIsLoading(true);//mostramos loading
    fetch('https://pablomagaz.com/api/posts')
      .then(response => response.json())
      .then(data => setPosts(data.posts))
      .finally(() => setIsLoading(false));//ocultamos el loading
  }, []); //El array vacío es el estado inicial y el effect no se volverá a ejecutar cuand
o su contenido cambie

  const loading = (isLoading) ? Loading... : null;

  return (
   <fieldset>
      <div>
        { loading  }
        { posts.map((post, key) => (
          <div key={ key }>
            { post.title }
          </div>))}
      </div>
    </fieldset>);
```

```
}

export default UseEffectApiRequest;
```

USEREDUCER

Sí, en efecto, has leído bien. useReducer es probablemente uno de los hooks más sorprendentes y un claro guiño a Redux ya que nos ofrece la posibilidad de tener Redux en un solo hook y para ello useReducer dispone de state y dispatch, además de necesitar un reducer tradicional de Redux, es decir, una función de reducción que devuelve el siguiente state:

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

// definimos un valor inicial para el store
const initialState = { count: 0 };

//definimos un reducer que recibirá el state y el action
const reducer = (state, action) => {
  switch (action.type) {
    case 'increment': return { count: state.count + 1 };
    case 'decrement': return { count: state.count - 1 };
    case 'reset': return initialState
    default: return state;
  }
}

const UseReducerComponent = () => {
  //state es el valor del store, y dispatch nos permite disparar acciones
  const [state, dispatch] = useReducer(reducer, initialState);// Nuestro reducer y el valo
r inicial

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

export default UseReducerComponent;
```

Como vemos en el ejemplo, useReducer se encarga de que cada vez que hagamos un dispatch de una acción, el reducer que hemos definido recibirá tanto el state inicial, pasado por useReducer, como la acción disparada. Redux en un hook. Lógicamente

esto abre un nuevo debate sobre el futuro de Redux especialmente cuando lo combinamos con el Context Api y su versión en hook, useContext.

USECONTEXT

useContext es el hook encargado de gestionar la Context Api, pero basicamente context api nos permite crear un mecanismo (provider) para el paso de contextos en una jerarquia de componentes:

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

const UseContextComponent = () => {
  //Create context devuelve { Provider, Consumer }
  const CountContext = React.createContext(15);
  const count = useContext(CountContext);

  return (
    <div>
      <CountContext.Provider value={ count }>
        { count }
      </CountContext.Provider>
    </div>);
}

export default UseContextComponent;
```

USECALLBACK

La optimización ha sido otro de los focos con los nuevos hooks. useCallbackes el hook que permite disponer de memoria y evitar que una función se vuelva a ejecutar si sus parámetros no han cambiado, devolviendo el valor almacenado o memorizado. Esto es especialmente útil para operaciones que tienen un elevado coste computacional o queremos controlar si un componente se vuelve a renderizar o no, como hacemos hasta ahora con shouldComponentUpdate, según los parametros que recibe la función que lo devuelve:

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

const UseCallbackComponent = () => {
  const [text, setText] = useState('Hello!');

  // Función que va a ser memorizada
  const ChildComponent = ({ text }) => {
```

```
    console.log('rendered again!');
    return (
      <div>
        { text }
      </div>);
  }

   // Gracias a useCallback solo se renderiza cuando el valor indicado (text) cambie
   const MemoizedComponent = useCallback(, [text]);

   return (
     <div>
       <button onClick={() => setText('Hello!')}>Hello!</button>
       <button onClick={() => setText('Hola!')}>Hola!</button>
       { MemoizedComponent }
     </div>);
}

export default UseCallbackComponent;
```

USEMEMO

useMemo es una variante de useCallback y también como alternativa a shouldComponentUpdate. La única diferencia real es la sintaxis ya que su funcionamiento es el mismo:

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

//useCallback
const MemoizedComponent = useCallback(, [text]);

//useMemo
const MemoizedComponent = useMemo(() => , [text]);
```

USEREF

useRef es el hook para referencias del DOM pero podemos ver useRef también como un mecanismo para definir variables de instancia o incluso para permitirnos acceder a las props o state previos:

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

const UseRefComponent = () => {
  const [count, setCount] = useState(0);// 0 es el valor inicial
```

```
  // useRef se puede usar para alojar DOMS refs o cualquier tipo de referencia
  const prevCountRef = useRef();
  // Usamos useEffect para cambiar el valor de la referencia
  useEffect(() => {
    prevCountRef.current = count;
  });

  const prevCount = prevCountRef.current;

  return (
    <div>
      Count: { count } | Previous Count: { prevCount }
      <button onClick={() => setCount(count + 1)}>+</button>
      <button onClick={() => setCount(count - 1)}>-</button>
      <button onClick={() => setCount(0)}>Reset</button>
    </div>);
}

export default UseRefComponent;
```

Hemos visto los principales hooks, aunque existen alguno más como useLayoutEffect, que es la versión síncrona de useEffect o también useDebugValue que es un hook para debugging con React DevTools, no obstante, siempre es posible la creación de nuestros propios hooks, algo que veremos en próximos posts :)