Hooks Avanzados

Tal como avanzamos en la sesión 5 con los `states` vamos a profundizar más en los diferentes `Hooks` que nos ofrece React.

Los Hooks nos proporcionan una API más directa a los conceptos más utilizados de React, como los `props`, los `states`, el `contexto`, las `referencias` o el `ciclo de vida`, además de la capacidad de combinarlos.

Estos son completamente opcionales, compatibles con todas las versiones de React y reutilizables. Ya hemos visto el `useState`, por lo que vamos a tratar en esta sesión los `useEffect` para controlar nuestra aplicación, `useReducer` para controlar estados más complejos, `useContext` para compartir estado en nuestra app, `useRef` como medio para acceder al DOM y `useImperativeHandle` para personalizar el valor que se expone a los componentes usando ref.

useEffect

El Hook de efecto nos otorga la capacidad de disparar efectos secundarios desde un componente. Este hook recibe por parámetro una función que se ejecutará cada vez que nuestro componente se renderice, leyendo cualquier cambio de estado props nuevas o, simplemente, por ser la primera vez que renderiza nuestra aplicación.

Para usar este hook primero tenemos que importarlo, como viene siendo habitual, en nuestro componente desde la librería de React:

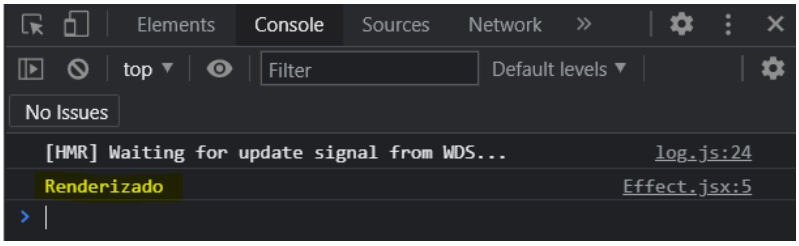
```
import React, { useEffect } from 'react'
```

Vamos a añadir un efecto en nuestro componente que se ejecutará cada vez que se renderice. Para ello vamos a ejecutar `useEffect` dentro del cuerpo de nuestro componente y le pasaremos por parámetro la función que queremos que se ejecute al renderizar:

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

function Effect() {
  useEffect(function () {
    console.log("Renderizado");
  });

  return <p>Así funciona useEffect</p>;
}

export default Effect;
```

Como podemos ver, cada vez que se renderice por primera vez nuestro componente se imprimirá por consola el mensaje `"Renderizado"`, funcionando de manera similar al ciclo `componentDidMount` que vimos en la sesión de `Fetch` pero de manera mucho más sencilla y simplificada.

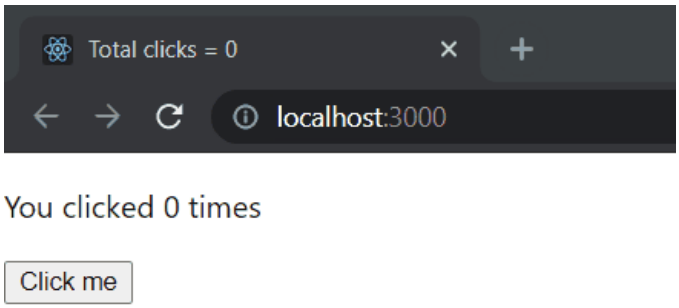
Así funciona useEffect



Ahora que hemos comprobado el funcionamiento de `useEffect` vamos a combinarlo con el hook `useState` y el ejemplo del contador de la sesión 5 para que nos actualice el título de la página con el número de veces que acumule el contador:

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

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

  useEffect(() => {
    document.title = `Total clicks = ${count}`
  })

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  )
};

export default Contador;
```

Con este hook estamos accediendo directamente al `document.title` y le estamos ordenando que nos indique en el mismo el numero acumulado en la variable `count` a tiempo real y teniendo en cuenta toda la información que nos está devolviendo el trabajo realizado por `useState`:



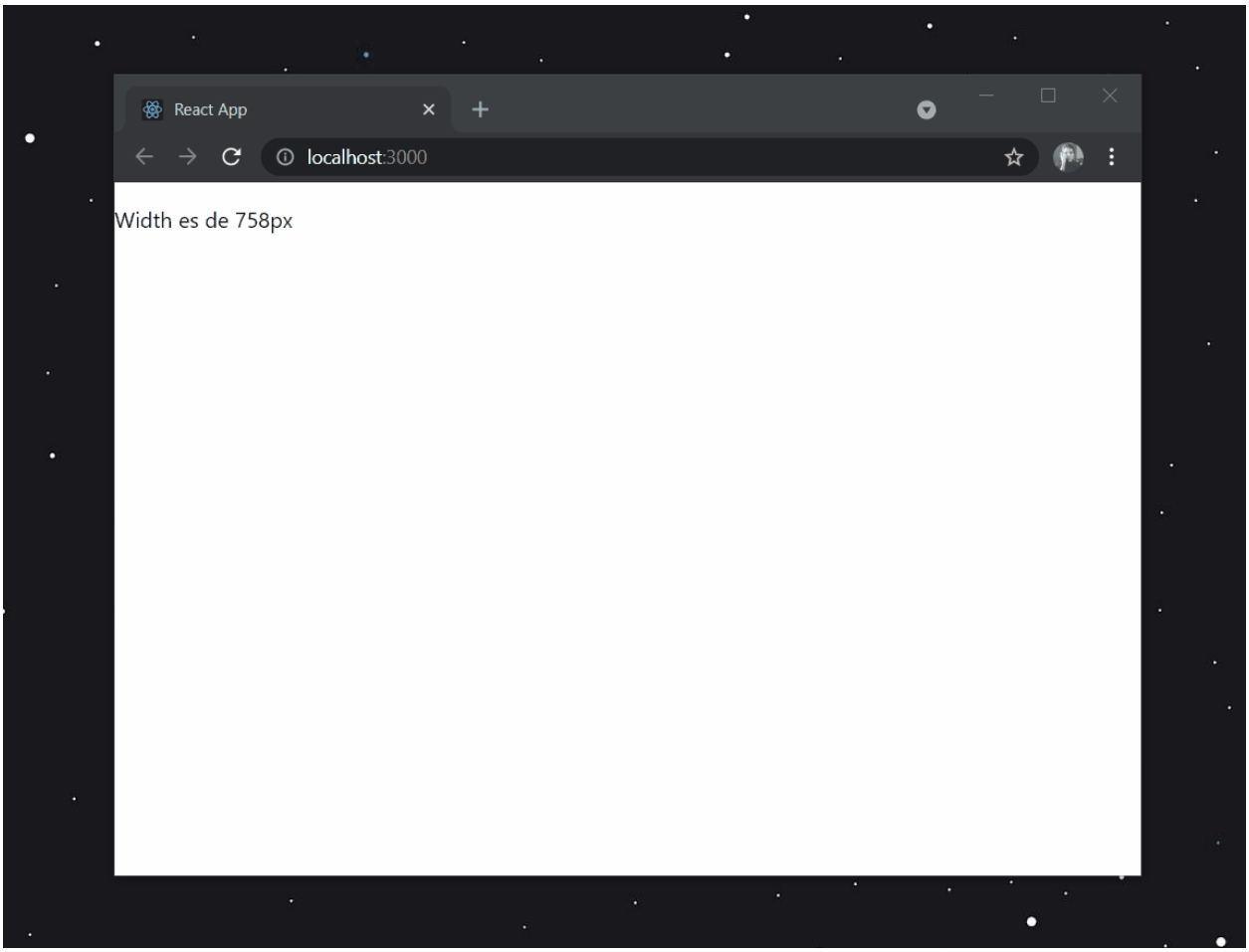
Es posible escuchar eventos del DOM suscribiéndonos al mismo, pero podemos provocar memory leaks si no tenemos en cuenta el desmontar dicha suscripción una vez ejecutada ya que por defecto `useEffect` se ejecuta cada vez que se realiza un nuevo renderizado.

Vamos a realizar algo no muy diferente al contador anterior que nos lea e indique el ancho de la ventana del navegador pasándole una segunda función que deje de escuchar a la suscripción para ahorrar recursos una vez este ejecutada:

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

function SizeListener() {
  const [width, setWidth] = useState(0);

  useEffect(() => {
    const updateWidth = () => {
      const width = document.body.clientWidth;
      console.log(`updateWidth con ${width}`);
      setWidth(width);
    };
    updateWidth();

    window.addEventListener("resize", updateWidth);

    return () => {
      window.removeEventListener("resize", updateWidth);
    };
  };

  return (
    <div>
      <p>Width es de {width}px</p>
    </div>
  );
```

```
}
export default SizeListener;
```

En este ejemplo hemos seteado a 0 nuestro `useState` para que se sobrescriba mediante `setWidth` con lo que nos lea la función en `document.body.clientWidth`. De esta forma al entrar en el return nos va a actualizar la información y una vez mostrada va a parar el `EventListener` hasta que volvamos a modificar su valor.



useReducer

El hook `useReducer` nos permite modificar estados más complejos que los vistos y utilizados a través de `useState`. Con este último tendríamos que dividir el estado en

muchos más pequeños para controlarlos, por lo que es recomendable utilizar `useReducer` ante grandes objetos y evitar generar fallos o conflictos en la aplicación.

`useReducer` hace uso del método de array `reducer`: una función de devolución de llamada que almacena cada elemento que le proporcionamos en un array y devuelve el valor final. Este método es uno de los más limpios y efectivos a la hora de iterar y procesar datos almacenados en un array.

En este caso recibirá el estado almacenado y la acción que lanzamos desde el componente. Esta `acción` se ejecutará mediante el comando `dispatch`. Con este bloque de código se entenderá mejor:

```
const reducer = (state, action) => {
...
// El estado se actualizará según lo que marque la acción
...
return updatedState;
}
```

Habitualmente la acción será un objeto `type` que incluya un `payload` o carga útil para actualizar un estado de alguna manera.

```
{type: "INCREMENTAR STATE POR PAYLOAD", payload: 5}
```

La esencia de todo esto es que, dependiendo del tipo de `action.type` se ejecutará un bloque de código u otro que devuelvan un nuevo estado.

Imaginemos que tenemos una aplicación de finanzas personales en la que tenemos un `INITIAL_STATE` como este:

```
const INITIAL_STATE = {
    moneyInBank: 0,
```

```
    moneyInSofa: 999,
    billsToPay: 1000
}
```

```
const reducer = (state, action) => {
    switch(action.type){
        case "INGRESAR DINERO EN EL BANCO":
            return {...state,
                    moneyInBank: state.moneyInBank + action.payload
                   };
        case "PAGAR FACTURAS":
            return {...state,
                    billsToPay: state.billsToPay - action.payload
                   };
        case "BUSCAR MONEDAS EN EL SOFA":
            return {...state,
                    moneyInBank: state.moneyInBank
                                 + state.moneyInSofa,
                    moneyInSofa: 0
                   };
  };
}
```

Cada `caso` de nuestro `switch` está devolviendo un objeto diferente, esto evita que cambiemos de `state`.

El hecho de que declaremos la función fuera de nuestro `useReducer` nos ayuda a mantener toda la lógica para actualizar el estado en un lugar ordenado. Ahora que tenemos claro como ordenar y preparar los casos y acciones nos falta una última función que ejecutará activar dichos cambios: el `dispatch`.

Siguiendo el ejemplo de antes, en el segundo caso el usuario debería tener la capacidad de pagar unas facturas por la cantidad que desee. En este caso podemos llamar al `dispatch` de la forma siguiente:

```
...
dispatch({type: "PAY SOME BILLS", payload: 100});
...
```

Por lo que si combinamos el código anterior con este podremos entender fácilmente que al ejecutar esta línea le va a restar `100` a través del `action.payload` al estado de las facturas por pagar, las cuales se inicializaron en `1000`.

Con solo 3 cosas: `reducer`, `action` y `dispatch`, podemos controlar estados varios sin necesitad de crear 3 `useState` diferentes, permitiéndonos crear aplicaciones mucho más complejas sin crear montañas de código.

useContext

`useContext` nos va a permitir manipular el contexto de nuestros componentes y comunicarlos entre si. Es decir, imaginad que desde un componente pueden preparar, consultar y consumir lo que hay en otro componente.

Vamos a crear un ejemplo con un intercambiador de temas para nuestra aplicación. En esta ocasión vamos a usar bootstrap y sass para un mejor y eficaz acabado, pero se puede utilizar cualquier forma de estilizar dichos temas. Si queréis usar bootstrap tendréis que instalarlo en el proyecto con el siguiente código: `npm install react-bootstrap@next bootstrap@5.1.0`

Para empezar tendremos que crear una carpeta llamada `context` en nuestro `src`, al igual que hicimos con los componentes. En ella insertaremos un nuevo archivo llamado `ThemeContext.jsx` en el que crearemos el contexto:

```
import React from "react"

const ThemeContext = React.createContext({
  theme: "light",
  setTheme: () => {},
})

export default ThemeContext
```

La función `React.createContext()` se utiliza para crear el contexto, en nuestro caso necesitamos una variable `theme` con el valor del tema actual: "light". Y para cuando queramos actualizar esta variable definiremos `setTheme`.

Ahora que tenemos el contexto preparado vamos a crear un componente `Header` de la misma forma que hemos creado prácticamente todos anterioremente:

```
import React, { useContext } from 'react'
import { Form, Navbar, Nav, FormControl, Button } from 'react-bootstrap'
import ThemeContext from '../contexts/ThemeContext'


const Header = () => {
  const { theme } = useContext(ThemeContext)

  return (
    <>
      <Navbar bg={theme} variant={theme}>
        <Navbar.Brand href="#home">Logo</Navbar.Brand>
        <Nav className="mr-auto">
          <Nav.Link href="#home">Home</Nav.Link>
        </Nav>
        <Form inline>
          <FormControl type="text" placeholder="Search" className="mr-sm-2" />
          <Button variant="outline-info" className="search-button">Search</Button>
        </Form>
      </Navbar>
    </>
  )
}

export default Header
```

En este caso hemos usado `useContext` y `ThemeContext` para recuperar el contexto del tema y la variable `theme` con el tema actual que se reflejará en nuestra aplicación.

También hemos creado un componente `ThemeSwitcher` que contendrá la lógica para cambiar entre temas de la siguiente forma:

```
const ThemeSwitcher = () => {
  const { theme, setTheme } = useContext(ThemeContext)

  return (
    <Button
      onClick={() => setTheme(theme == "dark" ? "light" : "dark")}
      className="button-theme"
    >
      <img src={theme == "dark" ? Sun : Moon} className="theme-icon" alt="theme" />
    </Button>
  )
}
```

De esta manera y usando unas imágenes del Sol y la Luna, por ejemplo, almacenadas en nuestra aplicación podemos determinar que si no está activada una se reflejará la otra y viceversa mediante un sencillo ternario.

En los estilos `.scss` que hemos mencionado anteriormente tendremos que definir la clase `dark` y la clase `light` que nos cambiará en este caso tanto el color del fondo como del contenido:

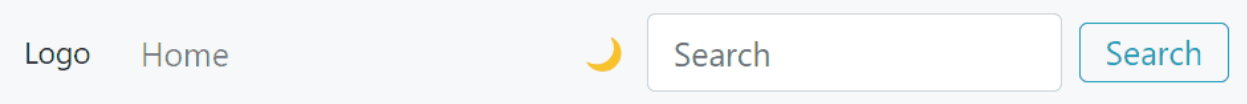
```
.dark {
  background-color: #404042;
  color: gray;
}
```

Todos estos componentes que estamos creando y, en los cuales no vamos a entrar en detalle se pueden modificar y ajustar tanto en la hoja de estilos como en los componentes en si. En nuestro caso vamos a añadirle al `NavBar` de bootstrap el botón que componentizamos hace un par de pasos y que seteará el tema a `dark` o `light` al hacer click en él:

```
import React, { useContext } from 'react'
import { Form, Navbar, Nav, FormControl, Button } from 'react-bootstrap'
import ThemeContext from '../contexts/ThemeContext'
import ThemeSwitcher from './ThemeSwitcher'
```

```
const Header = () => {
  const { theme } = useContext(ThemeContext)

  return (
    <>
      <Navbar bg={theme} variant={theme}>
        <Navbar.Brand href="#home">Logo</Navbar.Brand>
        <Nav className="mr-auto">
          <Nav.Link href="#home">Home</Nav.Link>
        </Nav>
        <ThemeSwitcher />
        <Form inline>
          <FormControl type="text" placeholder="Search" className="mr-sm-2" />
          <Button variant="outline-info" className="search-button">Search</Button>
        </Form>
      </Navbar>
    </>
  )
}

export default Header
```

De esta forma tenemos un botón que al ejecutar `onClick` nos va a renderizar un tema u otro gracias al que el contexto de la aplicación está cambiando:



`useContext` es muy útil a la hora de incluir varios temas, idiomas o mostrar una determinada información según el usuario que esté logueado en nuestra aplicación.

useRef

Este hook nos va a permitir trabajar con referencias en nuestros componentes. `useRef` nos retorna un objeto con una propiedad inmutable llamada `current` cuyo valor persistirá durante los renderizados y ciclos de vida.

`useRef` se suele utilizar mucho a la hora de usar referencias al DOM o a otros componentes de nuestra aplicación.

Vamos a ver su comportamiento en este sencillo ejemplo:

```
function Counter() {
  const [count, setCount] = useState(0)
  const prevCount = useRef(0)
  useEffect(() => {
    // Esto se ejecuta cada vez que el componente se renderiza
    prevCount.current = count
  })
  return (
    <div className="flex flex-col justify-center items-center m-8">
      <div>
        Prev: {prevCount.current}, Count: {count}
      </div>
      <button className="bg-purple-700 px-6 text-white rounded-md hover:bg-purple-500"
        onClick={() => setCount((c) => c + 1)}>+1</button>
    </div>
  )
}
```

En nuestro código podemos observar como al guardar nuestro estado previo en `useEffect` no queremos re-renderizar el componente. Por lo tanto, para no hacer un bucle infinito estamos utilizando `useRef` mediante la constante `prevCount`.

Esta constante está recogiendo el valor de `count` mediante `.current` y manteniendo el valor previo recogido por el `setCount`. La referencia de `prevCount` es nada más y nada menos que el valor almacenado justo antes de re-renderizar, en el cual persiste la información de forma inmutable hasta que se vuelva a renderizar, cosa que no hace la variable `count`.

useImperativeHandle

El hook `useImperativeHandle` nos permite pasar valores y funciones de componentes hijos de vuelta al padre haciendo uso de `ref`. En ese punto, el componente padre podrá hacer uso de ellos e incluso pasarlo a otro hijo.

Hay que tener en cuenta de que únicamente podremos pasar un `ref` como prop a un componente hijo siempre y cuando envolvamos a este componente en un `forwardRef`. Como el propio termino indica, esta herramienta nos permitirá pasar hacía "delante" una `ref`.

Vamos a ver esto mejor en el siguiente ejemplo:

```
import { useState, useRef, forwardRef, useImperativeHandle } from "react";

const ChildOne = forwardRef((props, ref) => {
  const [count, setCount] = useState(0);

  useImperativeHandle(ref, () => ({
    count,
  }));

  const updateCount = () => {
    setCount((c) => c + 1);
    console.log(count + 1);
  };

  return <button onClick={updateCount}>Increment</button>;
});

const ChildTwo = forwardRef((props, ref) => {
  const checkCount = () => console.log("->", ref.current.count);

  return <button onClick={checkCount}>Count</button>;
});

const Parent = () => {
  const ref = useRef();

  return (
    <div>
      <ChildOne ref={ref} />
      <ChildTwo ref={ref} />
    </div>
  );
};
```

```
export default Parent;
```

Como podéis ver no es más que otro contador en el que damos valor a `ref` en el componente padre y se lo pasamos a los dos componentes hijos.

Ambos componentes hijos están retornando a través de `forwardRef` para poder tomar `ref` como prop. El primer argumento que recoge `forwardRef` es una prop regular y el segundo el `ref`.

Si vamos de arriba abajo vemos que `ChildOne` tiene su propio estado interno, el cual está siendo controlado mediante `useState` y tiene un método para actualizar el valor de `count`. El valor de `count` está siendo accesible para el componente padre gracias al uso de `useImperativeHandle`, el cual recoge `ref` como argumento y retorna un objeto con la variable `count`.

El componente padre puede acceder ahora a `count` y pasarla como prop al componente hijo `ChildTwo`, el cual reflejará el consola el valor de `ref.current.count` cada vez que se haga click en el. De esta manera podemos ver el flujo de pasar valores de vuelta a componentes padre y, una vez hecho, devolverlo a un componente hijo diferente.

Como hemos visto antes, gracias a `useRef`, no se está re-renderizando nada en el componente padre.