**­­­­­GifExpertApp**

En esta nueva sección veremos la comunicación entre componentes, cómo es el flujo de intercambio de información, reforzamiento sobre los Proptypes, cómo hacer peticiones a una API con un hook…

Lo primero que debemos hacer, como siempre, es crear una nueva aplicación: npx create-react-app nombre\_aplicacion

Para esta aplicación vamos a hacer uso de Giphy, por lo que necesitamos un API key.

Una vez la tenemos, nos vamos a Docs. Vamos a trabajar con el Search Endpoint. Podemos ver toda la documentación necesaria ahí.

1. **Creación de listas**

Supongamos que tenemos un array y queremos mostrar dicho array en una lista ordenada. Tendríamos que recorrer cada uno de los elementos de nuestro array, y devolverlos para imprimirlos. Podríamos pensar que lo podemos hacer con un for each, pero esto no nos devuelve nada.

Podemos utilizar algunos métodos que tienen los arrays, como por ejemplo el método .map(). Si nada más que devolvermos los elementos nos dará un error: Cada uno de los elementos hijos de una lista debe tener una clave única, una property *key*

El método .map recibe dos argumentos: un elemento del array, y su índice, y podríamos pensar que podemos usar dicho índice para asignárselo a la key. Esa key, sirve para que react sepa sobre qué elemento está iterando, y si cambia o se tienen que hacer renderizaciones o borrar algo, se va a basar en el key. Si usáramos el índice, podría ser que borráramos por ejemplo el primer elemento pero ese primer elemento se mantendría y simplemente se cambiaría el último, teniendo resultados nos esperados.

Para solucionar esto, el key va a ser el mismo que el valor del elemento que vamos a imprimir.

Usualmente usaremos el valor único del id de cada uno de los elementos. Por ahora, la cosa nos va a quedar así:

<ol>

                {

*categories*.map( *category* **=>** {

                        return <li key={ *category* }> { *category* } </li>;

                    } )

                }

            </ol>

Esto funciona, pero ahora se nos presenta un nuevo problema: Si queremos añadir un nuevo elemento a nuestro array, este será una constante, y no pueden mutar.

Tenemos que tratar de decirle a React cómo hacer la nueva renderización de nuestra lista. Para ello, vamos a hacer uso de nuevo del hook de *useState.*

**const** [ *categories*, setCategories ] = useState( [ 'One Punch', 'Samurai X', 'Dragon Ball'] );

Hacemos la asignación desestructurada, en la que el primer argumento será el array de nuestro valor inicial, el que le pasamos al *useState*, y el segundo argumento será el método para cambiar el estado. Vamos a crear entonces una función que nos permita hacer estas modificaciones:

**const** handleAdd = () **=>** {

*//setCategories( [...categories, 'Prueba']);*

        setCategories( *cats* **=>** [...*categories*, 'Prueba']);

    }

Cuando usamos el *setCategories*, estamos sobreescribiendo el estado actual, con lo que no podríamos simplemente poner en dicho método el nuevo elemento que queremos agregar.

Podemos hacerlo de dos formas: En primer lugar hacemos uso del operado *spread*, y creamos un nuevo array con los elementos del array anterior y con el nuevo elemento que queremos añadir.

La segunda solución es haciendo uso del callback que puede recibir el *setCategories*. A dicho callback le pasamos el arreglo actual, y devolvemos el nuevo array con el elemento que acabamos de añadir. Más adelante veremos que esta forma es muy útil.

1. **Comunicación entre componentes**

Vamos a crearnos un nuevo componente llamado AddCategory en el que vamos a implementar la funcionalidad de añadir a nuestra lista de categorías una nueva.

Vamos a crear un *input* de tipo *text* para que podamos escribir las categorías que queremos añadir.

Esta caja de texto necesitará algún estado para que sepamos lo que el usuario introduce. Para ello, usaremos un *useState* de nuevo, al que le pasaremos un valor inicial que tendrá nuestra caja de texto.

Dentro de las propiedades del input, pondremos que el *value* será igual al valor inicial, *inputValue*.

**const** [ *inputValue*, setInputValue ] = useState('Hola mundo');

Veremos en nuestra página que en la caja de texto aparece el valor indicado, pero si intentamos escribir algo no nos lo va a permitir. Para ello debemos manejar el *onChange* de la propiedad *value*. En dicho método podemos poner un callback para manejarlo.

Vamos a crearnos una función, *handleInputChange* que va a recibir el valor del campo, y vamos a trabajar con la propiedad *target* para poder extraer el valor introducir y mostrarlo:

**const** handleInputChange = ( *e* ) **=>** {

        console.log(setInputValue(*e*.*target*.*value*));

    }

Nótese que hemos de llamar al setInputValue para que pueda cambiar el valor de nuestra caja de texto.

También vamos a querer manejar esto de tal forma que cuando escribamos y le demos al enter, se dispare un evento. Para ello vamos a crear un formulario, y nos vamos a crear una función handleSubmit que sea la que se dispare cuando intentemos enviar el formulario.

Si lo dejásemos tal cual, al presionar al enter, hace el posteo del formulario y recarga por completo. Esta práctica dejó de hacerse hace tiempo. Para evitar ese comportamiento, hemos de usar la propiedad *preventDefault()* del evento.

El componente al final nos quedará tal que así:

**const** handleSubmit = (*e*) **=>** {

*e*.preventDefault();

        console.log('Submit hecho');

    }

    return (

            <form onSubmit={ handleSubmit }>

                <input

                    type="text"

                    value={ *inputValue* }

                    onChange = { handleInputChange }

                />

            </form>

    )

Démonos cuenta de que en la devolución de la función, hemos metido todo dentro de la etiqueta del formulario, sin necesidad de meterlo en un div o en un fragment.

Como vemos, hemos puesto que al hacer el submit del formulario se dispare la función handleSubmit, la cual previene el comportamiento por defecto del formulario, y nos imprime un mensaje diciendo que se ha hecho el submit.

A continuación lo que querremos hacer es comunicar este fragmento con el fragmento que usábamos para añadir nuevos elementos a una lista. Pero, ¿cómo hacemos para comunicar los componentes? Pues tenemos varias formas.

Como sabemos, en las etiquetas de nuestras categorías podemos añadir propiedades, pero en esta ocasión vamos a mandarle una función. La función que mandaremos será setCategories, una de las funciones obtenidas del uso del useState. Vamos a ponerle el nombre que queramos a la propiedad (debería ser relacionado con lo que mandamos) y a esa propiedad le asignaremos la referencia a la función.

<AddCategory setCategories = { setCategories }></AddCategory>

En este caso le pusimos el mismo nombre que la función. Ahora, para usar esa función en el componente, tenemos que obtenerla de las *props.* Ahora, para llamar a la función usaremos la siguiente expresión.

setCategories( *cats* **=>** [...*cats*, *inputValue* ] );

Lo haremos así porque no tenemos el acceso a las categorías porque no se lo pasamos como argumento. Recordemos que el setCategories recibe un callback que tiene el estado anterior.

Por ahora va bien la cosa, pero tenemos un error de validación, y es que, si en nuestra caja de texto le damos a intro cuando está vacía, eso funciona y lo añade, lo cual no debería ser válido. Tenemos que hacer una pequeña validación. Vamos a meter la llamada a la función dentro de un if. Vamos a quitarle los espacios en blanco a los lados de la cadena y pediremos que la longitud de esta sea >2.

**const** handleSubmit = (*e*) **=>** {

*e*.preventDefault();

        if( *inputValue*.trim().length > 2){

            setCategories( *cats* **=>** [...*cats*, *inputValue* ] );

            setInputValue('');

        }

    }

También hemos reseteado el valor para que no pueda hacer un doble posteo.

Antes de terminar esta parte, no temos el siguiente error: Si en nuestra caja de texto no ponemos un valor inicial, al escribir algo nos marcará un error. Eso es porque inicialmente, el estado inicial es undefined, lo cual cambiaría si introducimos algo. La solución es sencilla: En lugar de no pasarle nada como estado inicial, pondremos una cadena vacía.

**const** [ *inputValue*, setInputValue ] = useState('');

\*\*\*

OJO: Si queremos establecer que la función que le pasamos a nuestro componente sea obligatoria, lo tenemos que poner así:

*AddCategory*.*propTypes* = {

*setCategories* : *PropTypes*.*func*.*isRequired*

}

Nótese que al lado de PropTypes hemos tenido que añadir *.func*

1. **Fetch hacia una API**

Lo que vamos a querer ahora es crear un nuevo componente que conforme introduzcamos un nuevo elemento, haga las peticiones http y nos traiga las imágenes correspondiente a dicho elemento y se despliegue en nuestra página.

A nuestro componente vamos a pasarle una propiedad, que será cada una de las categorías que queremos mostrar.

Ahora en nuestro componente principal realizaremos un mapeo de las categorías que tenemos, y en el callback llamaremos a este nuevo componente pasándole cada una de las categorías:

*categories*.map( *category* **=>**

                        <GifGrid

                            key={*category*}

                            category={ *category* }></GifGrid>)

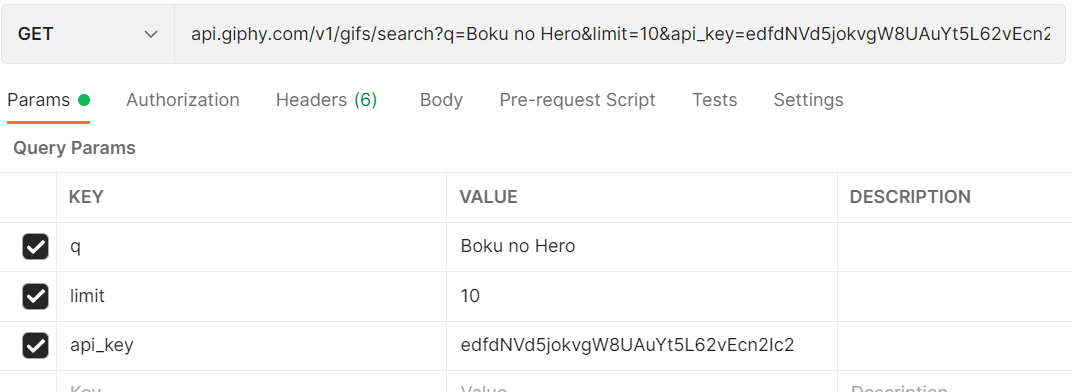
Volvamos a nuestro componente. Queremos hacer peticiones http con las categorías recibidas. Para empezar, nos vamos a la aplicación GIPHY, a la sección EndPoints -> Search, y copíamos la url del GIF.

Abrimos postman, y realizamos la petición http. Si hacemos esto tal cual, no va a funcionar, pues nos falta la api\_key de nuestra app. Hemos de pasárselo como parámetro a la url. Podemos hacerlo de dos formas:

1. Al final de la url ponemos ?api\_key= y le asignamos la url
2. En el apartado de parámetros de postman, ponemos la api\_key

Además, añadiremos otro parámetro más, q (de query) en el que le pasaremos la petición de lo que queramos buscar.

Esta petición nos va a traer una colección de muchas imágenes que coinciden con la petición. Podemos ajustar esta búsqueda, poniendo otro parámetro, *limit*, para que nos traiga sólo 10 imágenes.



Copiamos ahora la url, y creamos una constante a la que le asignemos dicha url.

Ahora vamos a utilizar el fetch, al que le pasaremos dicha url. También vamos a poner nuestra función para obtener gif como asíncrona. Lo siguiente será crear una constante data, que la vamos a recibir en formato json.

**const** *url* = "https://api.giphy.com/v1/gifs/search?q=Boku+no+Hero&limit=10&api\_key=edfdNVd5jokvgW8UAuYt5L62vEcn2Ic2";

**const** *respuesta* = await fetch( *url* );

**const** {*data*} = await *respuesta*.json();

Ahora tenemos que procesar la data obtenida, para sacar lo que queremos.

**const** *gifs* = *data*.map( *img* **=>** {

            return{

*id*: *img*.*id*,

*title*: *img*.*title*,

*url*: *img*.*images*?.*downsized\_medium*.*url*

            }

        })

Nótese que en la url hemos puesto el signo de interrogación: Esto es para que sólo utilice dicha url si viene la imagen.

En principio todo funciona bien, pero vamos a tener un error. Para verlo, vamos a hacer uso del *useState*.

Imaginemos que creamos un contador, y un botón que cada vez que lo pulsamos incremente dicho contador. Nos daremos cuenta de que cada vez que pulsemos se va a realizar la petición http. Cada vez que hay un cambio en el componente se dispararía. Cuando react detecta un cambio, tiene que actualizar las referencias y ejecuta todo el código del componente, y como la llamada a la función que hace la petición está al mismo nivel que el resto del código del componente, volverá a hacer la llamada cada vez que se renderice. La solución para esto es hacer uso de un nuevo hook: *useEffect()*. Esto nos va a permitir ejecutar cierto código de manera condicional. Este hook recibe dos argumentos: El primero será la función que queremos ejecutar, y el segundo será un array de dependencias. Si le mandamos un array vacío, el useEffect se disparará una única vez, cuando se renderice el componente por primera vez.

1. **Crear listados**

Vamos a crearnos un nuevo estado, cuyos estados van a ser los gif que vamos a recibir. Lo que queremos hacer es crearnos un listado con dichos gifs.

Nos vamos a crear un nuevo componente llamado *GifGridItem*. Este componente va a recibir una imagen/gif. En este componente vamos a hacer la asignación desestructurada para sacar el título del gif y su url. Nuestro componente va a quedar de la siguiente forma.

export **const** GifGridItem = ( {*title*, *url*} ) **=>** {

    console.log( *title*, *url* );

    return (

        <div className='card animate\_\_animated animate\_\_fadeIn'>

            <img src={*url*} alt={*title*}/>

            <p>{*title*}</p>

        </div>

  )

}

Después, en nuestro componente GifGrid vamos a llamar a este componente tal que así:

*images*?.map( *img* **=>** (

                    <GifGridItem

                        key = {*img*.*id*}

                        { ...*img*}

                        />

                ))

Así estamos recorriendo todos los gifs obtenidos, y a nuestro nuevo componente le estamos dando el id de los gifs, y usamos el operador spread para poder mandarle las propiedades de nuestros gifs.

1. **Helpers - getGif**

Hasta lo de ahora cuando lanzamos nuestra aplicación siempre nos devuelve los mismos gif cuando hacemos una petición. Eso es porque la url que le pasamos es siempre la misma. Tenemos que mandarle como parámetro a esa url la categoría que queremos que nos muestre.

**const** *url* = `https://api.giphy.com/v1/gifs/search?q=${ encodeURI(*category*) }&limit=10&api\_key=edfdNVd5jokvgW8UAuYt5L62vEcn2Ic2`;

Nótese que le hemos pasado la categoría dentro de una función. Esto lo que hace es eliminar los posibles espacios en la cadena de la categoría para que no llegue a dar problemas.

Podemos extraer dicha funcionalidad en un nuevo componente para que el componente en que lo tenemos quede un poco más limpio. Dicho componente va a ser *getGifs*, el cual vamos a guardarlo en una carpeta *Helpers*. Veamos cómo va a quedar nuestro nuevo componente.

export **const** getGifs = **async** ( *category* ) **=>** {

**const** *url* = `https://api.giphy.com/v1/gifs/search?q=${ encodeURI(*category*) }&limit=10&api\_key=edfdNVd5jokvgW8UAuYt5L62vEcn2Ic2`;

**const** *respuesta* = await fetch( *url* );

**const** {*data*} = await *respuesta*.json();

**const** *gifs* = *data*.map( *img* **=>** {

        return{

*id*: *img*.*id*,

*title*: *img*.*title*,

*url*: *img*.*images*?.*downsized\_medium*.*url*

        }

    })

    console.log(*gifs*);

    return *gifs* ;

}

Como vemos, le estamos pasando desde el componente padre la categoría que queremos que nos muestre, y hacemos que la función nos devuelva el resultado de la petición. Como se puede ver, tenemos que exportar nuestra función.

OJO, que como la función es asíncrona, nos devuelve una promesa, así que en el componente padre lo tenemos que tratar como tal.

Ahora, si nos fijamos en la consola veremos que tenemos un error. Nos dice que react hook effect tiene una dependencia desaparecida: ‘category’.

Tenemos que pensar que si la categoría cambiar, entonces se tendría que volver a renderizar esa parte del formulario. Para solucionar esto, en el hook useEffect tenemos que pasarle como segundo argumento: category.

useEffect(() **=>** {

    getGifs(*category*).then((*setImages*);

}, [*category*]);

1. **useFetchGifs**

Vamos a hablar de los custom hooks. Estos no son más que una forma de extraer lógica de algún componente o lógica que queremos reutilizar de tal forma que sea sencillo usarlo de nuevo.

Vamos a crearnos un hook de tal forma de que cuando se haga la petición, nos diga que está cargando, y que cuanto terminó la carga nos lo indique también.

Para crear un nuevo hook, se recomienda que se haga en una nueva carpeta hooks.

En dicha carpeta vamos a crear un archivo *useFetchGifs*. El use significa que es un hook. No es obligatorio, pero sí muy recomendable.

Los hooks no son más que funciones. Estos hooks también pueden tener un estado, y podría indicarle a otros componente que lo utilicen cuándo renderizarlo cuando cambió. Los custom hooks funcionan como si fueran functional components. Pueden tener efectos, usar contextos…

export **const** useFetchGifs = (*category*) **=>** {

**const** [*state*, setState] = useState({

*data*: [],

*loading*: true,

  });

  setTimeout(() **=>** {

    setState({

data : [1, 2, 3, 4, 5, 6, 7],

loading: false

})

}, 3000 );

return state;

};

Cuando pasamos este hook, al principio pasamos el loading como false, y pasados los 3 segundos, cambia a true. Esto lo podemos usar para mostrar un mensaje diciendo que está cargando, y pasados esos 3 segundos, cambiará el mensaje diciendo que ya se ha cargado.

Vamos a usar este useFetchGifs para que se encargue de hacer la petición http cuando se cargue por primera vez el componente, o cuando cambie la categoría.

Perfectamente podemos añadirle efectos a nuestros hooks. Vamos a añadirle a nuestro anterior hook el useEffect para que nuestra petición http sólo se ejecute cuando sea necesario. Cabe resaltar que los efectos no pueden ser async, pues esperan algo síncrono.