



React hooks - Challenge!

Los hooks son funciones que permiten enganchar nuestros componentes funcionales a características propias de un componente de clase. Es decir, proporcionan un estado y un ciclo de vida a estos componentes evitándonos a los desarrolladores el uso de las clases.

Antes de ver ejemplos detallados de los hooks creamos un proyecto de React:

```
npx create-react-app example-hooks
```

Lo primero cambiamos nuestro APP:

```
import React from 'react';  
import './App.scss';  
  
const App = () => {  
  return (  

```

```
<div className="app">
  {/* Ejemplo de comentario en JSX... */}
</div>
);
}

export default App;
```

El hook de estado: useState

El hook **useState** es el que nos **permite agregarle un estado local** a un componente y **cambiar ese estado**. Como **primer parámetro** tendremos el **valor inicial del State** y como **segundo** el **método que modifica ese State**.

Veamos un ejemplo, creamos el componente ShowContent:

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

function ShowContent() {

  const [show, isShow] = useState(false);

  return (
    <div>
      <button
        type="button"
        onClick={() => isShow(true)}
      >
        Mostrar Contenido
      </button>
      {
        show && '¡Vamos Upgraders!'
      }
    </div>
  );
}

export default ShowContent;
```

Y para usarlo lo importaremos en el App:

```
import React from 'react';
import './App.css';

import ShowContent from './components/showContent';

const App = () => {
  return (
    <div className="app">
      <ShowContent />
    </div>
  );
}

export default App;
```

Vamos a hacerlo más divertido, vamos a pasarle un prop y que nos pinte el mensaje del prop una vez cambiemos el State del componente. En el App:

```
import React from 'react';
import './App.css';

import ShowContent from './components/showContent';

const App = () => {
  return (
    <div className="app">
      <ShowContent message="hola upgraders"/>
    </div>
  );
}

export default App;
```

Y para recibir esa prop, en showContent:

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

function ShowContent(props) {

  const [show, isShow] = useState(false);

  return (
    <div>
      <button
        type="button"
        onClick={() => isShow(true)}
      >
        Mostrar Contenido
      </button>
      { show && props.message }
    </div>
  );
}

export default ShowContent;
```

Antes de continuar vamos a desgranar un poco este componente ShowContent:

- El componente ShowContent tiene un botón → Mostrar Contenido.
- El componente ShowContent tiene una etiqueta → props.message.
- Tenemos la variable show que su estado inicial es false → show - false.
- Tenemos un evento onClick → modifica el estado inicial y lo cambia a true.
- Lo pintamos con un renderizado condicional → { show && props.message }.

¡BEER TIME!

<https://giphy.com/embed/zXubYhkWFc9uE>

Vamos a realizar un contador que comience la cuenta en 5 y que cada vez que clickemos en un botón se decremente una unidad. Además cuando lleguemos al 0 mostraremos un mensaje diciendo "La cuenta atrás a terminado - Beer time". Os indico paso a paso:

- Creamos un componente que se llame BeerCounterTime.
- BeerCounterTime tendrá un estado inicial de 5.
- Tendremos una función que decrementa la unidad.
- Hacemos un renderizado condicional para pintar nuestro mensaje.
- Extra! El mensaje que queremos renderizar se lo pasaremos por props.

El hook de estado: useEffect

Este hook **se ejecuta** siempre **después del primer renderizado** y después de cada **actualización** y, por lo tanto, se **utiliza** para **ejecutar funciones después de hacer render**. **useEffect recibe una función** que puede realizar todo tipo de operación incluyendo efectos secundarios. Un ejemplo típico de una operación que podemos querer realizar es una llamada a un servicio.

Un poco lío, verdad? vamos a verlo a través de un ejemplo y luego lo desgranamos poco a poco. Creamos un componente FormHeroes:

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

function FormHeroes() {

  const [name, setName] = useState('');
  const [heroName, setHeroeName] = useState('');

  useEffect(() => {
    console.log('hook -> NAME:', name);
  });
}
```

```

}, [name]));

useEffect(() => {
  console.log('hook -> LASTNAME: ', heroName);
}, [heroName]);

return (
  <div>
    <input value={name} onChange={(event) => setName(event.target.value)} />
    <input value={heroName} onChange={(event) => setHeroName(event.target.value)} />
  </div>
);
}

export default FormHeroes;

```

Y como siempre para usarlo lo que haremos es importarlo en el App:

```

import React from 'react';

// STYLES
import './App.css';

// COMPONENTS
import ShowContent from './components/showContent';
import FormHeroes from './components/formHeroes';

const App = () => {
  return (
    <div className="app">
      <ShowContent message="hola upgraders"/>
      <FormHeroes />
    </div>
  );
}

export default App;

```

Ahora si abrimos la consola e interactuamos con nuestros inputs veremos que cada interacción llama al useEffect ejecutando lo que tenemos dentro de este, una función de console.log.

¿si useEffect se ejecuta en cada renderizado cómo afecta esto al funcionamiento? ¿Cómo se puede evitar que entre en bucle?

Pasándole a la función un array como segundo parámetro. El valor de este array será el que React comparará para saber si tiene que volver a ejecutar el hook. En caso de no variar este valor, el hook no se ejecutará → En nuestro caso el segundo parametro es name y heroName:

```
useEffect(() => {  
  console.log('hook -> NAME:', name);  
}, [name]);
```

El hook useEffect simplemente realiza un console.log() en cada ocasión en que se ejecuta. Si **no le pasásemos el segundo parámetro [name]**, veríamos el resultado de **console.log()** cada vez que el usuario introduce un valor en **cualquiera de los dos inputs**. En cambio, al **pasarle la variable name** como dependencia, el hook se ejecuta solamente cuando el usuario introduce un valor en el **input correspondiente a name**.

Es el momento de ver paso a paso lo que hace nuestro FormHeroes:

- El componente FormHeroes tiene dos inputs → name / heroName.
- Los inputs en onChange → llaman a los setters de los valores iniciales de nuestro useState.
- Nuestros useState se inicializan vacíos → tienen un setter para cambiar su estado.
- El setter no solo cambia el estado sino que lo compara con el estado que había con anterioridad para ver si este ha sido modificado y lanzarse.

JoseMa Sing Star

<https://giphy.com/embed/11ahZZugJHrdLO>

Vamos a valorar las habilidades musicales de cada uno de los miembros del equipo de UpgradeHub, todos sabemos que el único crack es nuestro compañero José María. Por ello os voy a indicar lo que quiero que hagáis:

- Crear un componente JoseMaSinger.
- Este componente tendrá un input que cuando detecte que se ha introducido el nombre de nuestro querido José María nos mostrará el siguiente contenido : "¡Ha nacido una estrella!".
- De lo contrario quiero que mostréis un mensaje con el siguiente contenido: "Sigue intentándolo pero nunca llegarás a ser un buen JoseMa".
- Añade de forma elegante un mensaje para cada uno de tus compañeros.

El hook de contexto: useContext

Antes de ver el hook useContext debemos hablar brevemente de la API Context de React.

API Context nos permite acceder a datos globales de nuestra aplicación sin tener la necesidad de pasarle info manualmente. Simplemente lo tenemos en las props del componente que se

encuentra encapsulado. Un ejemplo típico son los datos de un usuario loggeado.

La **API de Context** contiene un componente **Provider** y un componente **Consumer**. El **primero** es el encargado de **propagar el contexto** a sus Consumers **hijos**, a través de una prop llamada *value*. El hook **useContext** es el **sustituto** del componente **Consumer**, haciendo mucho más sencillo el acceso y la suscripción a los cambios del value del Provider.

Veamos un ejemplo. En primer lugar creamos el contexto con los datos del usuario, con un objeto vacío como valor inicial:

```
import { createContext } from 'react';

const UserContext = createContext({});

export default UserContext;
```

Ahora vamos a crear un componente ShowUser:

```
import React from 'react';
import UserContext from './userContext';
import Button from './button';

const userMock = {
  name: 'Alberto',
  email: 'alberto.rivera@upgrade.com',
};

function ShowUser() {
  return (
    <UserContext.Provider value={userMock}>
      <Button />
    </UserContext.Provider>
  );
}
```

```
}  
  
export default ShowUser;
```

Después tendremos que crear nuestro componente Button que hemos definido con anterioridad:

```
import React, { useContext } from 'react';  
import UserContext from './userContext';  
  
function Button() {  
  const { name } = useContext(UserContext);  
  return (  
    <button type="button">  
      {name}  
    </button>  
  );  
}
```

¿Qué está pasando? pues simplemente estamos definiendo un contexto con un objeto vacío que se rellena dentro del componente ShowUser y como Button se encuentra dentro de este tiene acceso a través de las props.

Y de este modo nos comunicamos entre padres e hijos en React 🔥

Register Time

<https://giphy.com/embed/t0virGpgSlp4mkfiXg>

Es hora de empezar a hacer cositas más complejas con React, vamos a hacer un componente Register que cuando se rellene la info el usuario lo consuman todos los componentes que están dentro de este. Para ello:

- Creamos componente Register, User y Message.
- Creamos un RegisterContext → iniciado vacío.
- Cuando clickamos en submit de Register → User pintará la información solicitada (mínimo nombre y email). Y Message pintará el siguiente mensaje "Lo has logrado {user.name} eres un crack de React y de la vida 🍌"