DigitalHouse>
Coding School

Certified Tech Developer
The Ultimate Degree

Front End III

Ejemplos de map y keys

Map📌

Dijimos que **map** se llama desde un array y que retorna un nuevo array de resultados. Supongamos que tenemos un array de personajes de **Among Us** y queremos tener un array que nos diga cuáles personajes no eran el impostor y cuál sí era el impostor. El array tiene objetos literales con el nombre del personaje y una bandera (flag) que indica si es tripulante o impostor:

```
const tripulantes = [
   {nombre: "Mr. Poindibags", esImpostor: true},
   {nombre: "Bombom", esImpostor: false},
   {nombre: "Tito", esImpostor: false},
   {nombre: "X-Ray", esImpostor: false},
   {nombre: "Fixfox", esImpostor: false},
];
```

Nuestra función **map** podría simplemente utilizar la información de cada tripulante para crear una cadena (string) con una frase como: “Mr. Poindibags era el impostor” o “Fixfox no era el impostor”. Podríamos utilizar un lazo **for** para crear un nuevo array a partir del primero, o podríamos usar **map**. La implementación con el lazo **for** podría ser así:

```
// Creamos un nuevo array
let tripulacion = [];

// Iteramos sobre cada elemento en el array tripulantes
for(let i=0; i<tripulantes.length; ++i) {
   let tripulante =
    `${ tripulantes[i].nombre} ${tripulantes[i].esImpostor ? '' : 'no ' }era el impostor`;

   // Metemos cada tripulante en el nuevo array
   tripulacion.push(tripulante);
}
```

Por otro lado para hacer esto con **map**, debemos invocar la función sobre el array **tripulantes** y pasamos como primer argumento una función que toma cada elemento (en este caso lo nombramos **tripulante**) y usa una plantilla literal (template literal) para leer los datos del elemento **tripulante.nombre** y **tripulante.esImpostor** y crear la frase:

```
tripulante =>

      `${ tripulante.nombre} ${tripulante.esImpostor ? '' : 'no ' }era el impostor`
```

La función completa luce así:

```
let tripulacion = tripulantes.map(

   tripulante =>

      `${ tripulante.nombre} ${tripulante.esImpostor ? '' : 'no ' }era el impostor`

);
```

En cualquiera de los casos, el resultado será un array con los cinco elementos:

```
0: "Mr. Poindibags era el impostor"

1: "Bombom no era el impostor"

2: "Tito no era el impostor"
```

```
3: "X-Ray no era el impostor"

4: "Fixfox no era el impostor"
```

Como vemos, usar el lazo for es mucho más verboso, aunque muy eficiente, pero usar **map** nos permite escribir código más simple y es un buen ejemplo de programación funcional.

Keys 📌

Veamos el mismo ejemplo anterior, pero ahora con React. Supongamos que tenemos un componente de tipo **StatusTripulante** que muestra las frases y que deseamos formar una lista. Si usáramos un lazo **for**, tendríamos algo así:

```
const tripulantes = [
  {nombre: "Mr. Poindibags", esImpostor: true},
  {nombre: "Bombom", esImpostor: false},
  {nombre: "Tito", esImpostor: false},
  {nombre: "X-Ray", esImpostor: false},
  {nombre: "Fixfox", esImpostor: false},
];
// Creamos un nuevo array
let tripulacion = [];
for(let i=0; i<tripulantes.length; ++i) {
  let tripulante = <StatusTripulante {...tripulantes[i]}/>;
```

```
  tripulacion.push(tripulante);
}
lista = <ul>{ tripulacion }</ul>;
```

La expresión **{...tripulantes[i]}** lo que hace es expandir la información dentro de cada tripulante y pasarla como **props**. Como vemos, usar **for** produce un código verboso y más bien complejo.

En la industria se prefiere usar **map** de la siguiente manera:

```
const tripulantes = [
    {nombre: "Mr. Poindibags", esImpostor: true},
    {nombre: "Bombom", esImpostor: false},
    {nombre: "Tito", esImpostor: false},
    {nombre: "X-Ray", esImpostor: false},
    {nombre: "Fixfox", esImpostor: false},
];

let tripulacion = tripulantes.map(
    tripulante => <StatusTripulante {...tripulante}/>
);
```

```
lista = <ul>{ tripulacion }</ul>;
```

Nuevamente, la expresión **{...tripulante}** lo que hace es expandir la información dentro de cada tripulante y pasarla como **props**.

Como dijimos, si dejamos el código de esta manera recibiremos una advertencia sobre la falta de **keys**, pero no un error. ¿Qué sucede si hacemos caso omiso de la advertencia?

IMPORTANTE

Lo que puede suceder es una de dos cosas: podríamos tener un muy bajo rendimiento de la aplicación al hacer cambios en la lista, o podríamos tener resultados inesperados (bugs) dependiendo del tipo de operaciones que realicemos sobre la lista.

Esto ocurre porque cuando React aplica su algoritmo de diferenciación (diffing algorithm) sobre dos árboles del DOM, primero compara los dos elementos raíz, y el comportamiento es diferente dependiendo de los tipos de elementos raíz:

A. Siempre que los elementos raíz tengan diferentes tipos, React derribará el árbol actual y construirá el nuevo árbol desde cero.

B. Si los elementos raíz son del mismo tipo, React observará los atributos de ambos y mantendrá el mismo nodo DOM subyacente. Entonces, React solo actualizará los atributos modificados y enseguida recorrerá a los hijos de los elementos.

React recorre al nodo actual y al nodo nuevo al mismo tiempo y generará una mutación siempre que haya una diferencia entre los elementos.

Es en este punto donde se puede hacer una diferencia, porque si el usuario agrega, elimina, o actualiza algunos elementos, no se necesita recrearlos a todos, pero si React no identifica a cada uno de los elementos, los cambiará a todos en lugar de darse cuenta de que puede mantener intactos algunos, incluso quizá la mayoría.

Es por esto que React soporta el atributo **key**. Cuando los elementos de un mismo tipo dentro de un subárbol tienen **keys**, React las usa para hacer coincidir los elementos en el árbol original con los elementos del nuevo árbol. Esto es lo que le permite ahorrar cambios innecesarios.

¿Y qué podemos utilizar como key? Los atributos key pueden ser una propiedad ID que agreguemos, o alguna composición hecha con algunas partes del contenido del elemento. La key solo tiene que ser única entre hermanos, no globalmente. En nuestro ejemplo si tuviéramos un parámetro ID para nuestros tripulantes, nuestro código con **map** y **key** podría ser así:

```
let tripulacion = tripulantes.map(
    tripulante => <StatusTripulante key={tripulante.id} {...tripulante}/>
);

lista = <ul>{ tripulacion }</ul>;
```

¿Pero qué sucede en el caso de que no contemos con un ID para cada elemento y no haya una forma fácil de crear un atributo key? En nuestro caso, cada tripulante tiene solo dos datos: un nombre y un flag llamado **esImpostor**, no hay ID.
Recordemos cómo son nuestros elementos:

```
{nombre: "Mr. Poindibags", esImpostor: true}
```

No hay ID ni nada que podamos usar como tal, porque los nombres no tienen garantía de ser únicos.

Es en estos casos que podemos usar el parámetro index del elemento dentro del array como key. Esto lo logramos usando el argumento index de la función que pasamos a **map**:

```
let tripulacion = tripulantes.map(
    (tripulante, index) => <StatusTripulante key={index} {...tripulante}/>
);

lista = <ul>{ tripulacion }</ul>;
```

Usar el parámetro index del elemento dentro del array como **key** puede llegar a funcionar bien si los elementos de la lista nunca se reordenan, ya que los reordenamientos serán lentos. Además, tengamos en cuenta que cuando se utiliza el parámetro index del elemento dentro del array como key, los reordenamientos también pueden causar problemas con el estado de los componentes. Si el atributo key es el índice del elemento, entonces mover un elemento lo cambia a los ojos de React porque le modifica la key. Esto podría causar actualizaciones en formas inesperadas (bugs) en los componentes.

¡Hasta la próxima!