04-07a Introducción a la programación funcional en JavaScript

Qué es la programación funcional

Como hemos resaltado antes, se trata de un paradigma de programación; un estilo de construir la estructura y elementos de una aplicación, viendo la computación como la evaluación de funciones matemáticas, huyendo de aquellas prácticas que provocan cambios de estado y mutación de los datos.

Si no te has enterado es normal, acostumbrado a una única manera de programar, es complicado comprenderlo de forma teórica, pero vamos con un ejemplo práctico:

Así es como a la mayoría nos enseñaron a programar por defecto (de manera imperativa):

```
const numbers = [1, 2, 3, 4];
let doubled = [];

for(let i=0; i<numbers.length; i++){
  doubled.push(numbers[i]*2);
}

console.log(doubled) //==> [2, 4, 6, 8]
```

Así es como se haría de forma declarativa (usando programación funcional):

```
const numbers = [1, 2, 3, 4];
const doubled = numbers.map(n => n*2);

console.log(doubled) //==> [2, 4, 6, 8];
```

En ambos ejemplos nuestro objetivo era obtener el doble de cada valor de un array. En la primera versión imperativa, usamos un contador que es nuestro estado de recorrido del primer array para saber dónde insertar el doble de cada valor en el segundo array. ¿Cuántas veces habrá causado un bug ese maldito contador?.

Con la manera declarativa nos olvidamos del contador y nos centramos en especificar la función que queremos aplicar. Es decir, cuando usamos programación declarativa, las sentencias declaran QUÉ hacer, delegando en otra función, pero no CÓMO hacerlo, a diferencia de la programación imperativa. Interesante, ¿verdad?.

Herramientas en JavaScript

Si bien hay algunos lenguajes creados específicamente para ser usados de manera funcional, JavaScript no es uno de ellos. Pero podemos hacer uso de algunos conceptos, prácticas y librerías que nos permiten emplear este paradigma:

Asegurar inmutabilidad de los datos con los que trabaja tu aplicación.

- Inmutabilidad
- Usar funciones puras.
- Uso de funciones de orden superior.
- Uso del currying.
- Composición de funciones.

Este será tu maletín de herramientas para usar este paradigma de programación en JavaScript. A continuación vamos a definir, presentar librerías y mostrar ejemplos de cada una de ellas.

> En este post vamos a usar características de ECMAScript 6, si no está familizariado con esta versión de Javascript recomendasmo que veas este webinar y este post.

Inmutabilidad

Cuando usamos el paradigma funcional queremos siempre evitar modificar el valor de nuestras variables de forma colateral y trabajar con datos inmutables.

Veamos primero unos ejemplos para después formular la teoría. ¿Qué ocurre cuando modifico la copia de una variable?

```
let foo = 1;
let bar = foo;
bar+=1;

console.log(foo) // ==> 1
OK. El valor inicial de foo no se ve alterado. Tiene sentido. Sigamos...

let foo = "foo";
let bar = foo;
bar+="bar"

console.log(foo); // ==> "foo"
Igual que antes. Nada anormal...

let foo = [1, 2, 3];
let bar = foo;
bar.push(10000);

console.log(foo) // ==> [1, 2, 3, 10000]
```

El valor inicial de foo se modificó. Seguro que ya te has tropezado con fallos relacionados con esto (normalmente nuestras aplicaciones son mucho más complejas y hay partes en las que no se ve a simple vista la declaración de la variable) y es aquí donde se originan muchos bugs en JavaScript.

Cuando usamos el paradigma funcional queremos siempre evitar modificar el valor de nuestras variables y trabajar con datos inmutables. Para ello se utilizan las siguientes tácticas:

- Reducir / eliminar las asignaciones en la medida de lo posible. (Algunos lenguajes funcionales ni siquiera tienen operador de asignación '=')

- Usar estructuras de datos inmutables: Para esto existen librerías como immutable.js de Facebook y mori.

- Usar las funciones freeze y seal: freeze convierte un objeto en inmutable, de manera que no es posible cambiar las propiedades que tiene definidas, a menos que éstas sean objetos, y seal impide añadir nuevas propiedades pero permite reasignar el valor de las existentes (ojo si queremos 'congelar' todos lo niveles de propiedades de un objeto podemos usar helpers como deepfreeze).

- Hacer uso de librerías que ponen a disposición funciones que respetan el paradigma funcional como Ramda o Lodash/fp.

- Hacer uso de las funciones de copia como Array.slice, Array.from

Para el ejemplo del apartado anterior, podríamos evitar el "efecto colateral" de modificar la variable foo usando la función append de Ramda:

```
import { append } from 'ramda';
const foo = [1, 2, 3];
const bar = append(4, foo);

console.log(foo); // ==> [1, 2, 3]
```

Funciones puras

Una función pura es aquella cuyo resultado será siempre el mismo para un mismo valor de entrada, es decir, es determinista y sólo depende del argumento recibido; tiene transparencia referencial. Además no tiene efectos colaterales (no modifica ninguna variable global ni local).

Un ejemplo de función pura podría ser:

```
function double(n) {
  return n * 2;
}
```

Está funcion recibe un parametro de entrada 'n' que no se modifica (no muta), y devuelve siempre el mismo resultado para ese valor de entrada.

Algunos indicadores de que una función es impura son:

- No tiene argumentos de entrada.
- No devuelve ningún valor.
- Usa 'this'.
- Usa variables globales.

Ejemplos de funciones impuras son:

```
// Impura: Devuelve void y modifica el entorno.
console.log("Hola");

// Impura: No tiene argumento de entrada y su resultado es no determinista.
Math.random();

// Impura: Modifica el array como efecto colateral.
array.splice(2, 3);
```

El uso de funciones puras en lugar de funciones o métodos impuros nos asegura que nuestros datos no se vean alterados accidentalmente, además, como podrás apreciar mejor posteriormente, hace que nuestro código se descomponga en funciones más reutilizables y fáciles de testear, y tendrá una mayor legibilidad.

... Todo esto suena interesante pero muy teórico ¿ Podemos ver algún ejemplo sencillo en el que se aprecie la ventaja de utilizar funciones puras? Vamos a ello, imaginémos que queremos implementar un método que añada a un array la suma de sus elementos, es decir si tenemos un array que contiene [2,3] devolverá [2,3,5], ... parece fácil, buscamos en internet y con reduce podemos calcular la suma, y con push podemos añadir el elemento al array.

```
function appendSumOfValues(entryArray) {
  const total = entryArray.reduce((accummulator, currentValue) => accummulator + currentValue);
  entryArray.push(total);
    return entryArray;
}
```

Si llamamos a la función

```
const original = [3, 2]
console.log(appendSumOfValues(original));
```

Hasta aquí todo bien, ejecutamos y obtenemos por la consola el valor esperado: [3, 2, 5].

Vamos a probar un caso más:

```
const original = [3, 2]
console.log(appendSumOfValues(original));
console.log(appendSumOfValues(original));
console.log(appendSumOfValues(original));
```

¿ Qué esperaríamos como resultado?

```
[3, 2, 5]
[3, 2, 5]
[3, 2, 5]
```

¿ Qué obtenemos en realidad como resultado?

```
[3, 2, 5]
[3, 2, 5, 10]
[3, 2, 5, 10, 20]
```

¿! Comoooor !? ¿ Qué ha pasado aquí? El método push, modifica el array que le pasamos como parametro, esto hace que cada que llamemos a appendSumOfValues modique el array original (la función no es pura), ¿ Os imagináis este pufo en una función que calculara distintos tipos de descuentos en base a opciones de compra? :-)

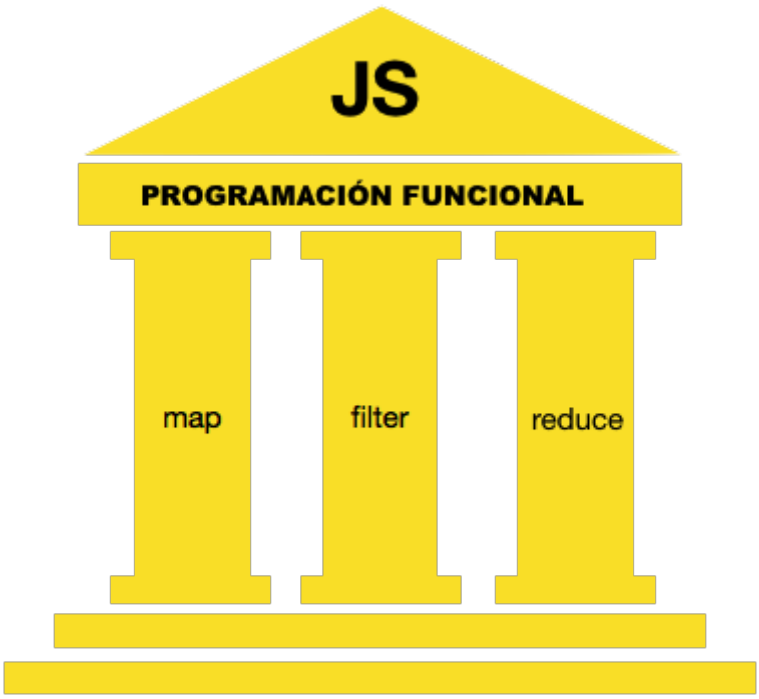
Ya veo, esto puede ser peligroso, ¿ Como puedo hacer está función pura? Usando un método para añadir un elemento a un array que cree una nueva estrcutura. Existen varias aproximaciones, en este caso elegiremos append.

```
function appendSumOfValues(entryArray) {
  const total = entryArray.reduce((accummulator, currentValue) => accummulator + currentValue);
  const result = entryArray.concat(total);
  return result;
}
```

Concat es un método que no muta el parametro de entrada, siempre devuelve un nuevo array, de esta manera conseguimos que appendSumOfValues sea una función pura y ahora si que obtendríamos el resultado esperado.

Funciones de orden superior

Las funciones de orden superior son aquellas que reciben una o más funciones como argumento o bien devuelven funciones como resultado. Nos interesan porque nos permiten reutilizar la forma de ejecutar otras funciones. En especial nos serán muy útiles map, filter y reduce, los pilares de la programación funcional en JavaScript, su interés principal está en usarlo sobre arrays.



- map: El resultado conserva la forma del argumento de entrada, probablemente con diferente tipo.
- filter: El resultado tendrá la misma forma que el argumento de entrada, probablemente de menor longitud.
- reduce: El resultado puede quedar totalmente transformado.

```
[1, 2, 3].map(n => n+1); // => [2, 3, 4]

[1, 2, 3].filter(n => n>1); // => [2, 3]

[1, 2, 3].reduce((acc, n) => acc + n, 0);
// 0 + 1 => 1
// 1 + 2 => 3
// 3 + 3 => 6
// => 6
```

Junto con estas funciones es interesante presentar el decorador unario. Veamos un ejemplo:

Vamos a pasar un array de strings a un array de números, para eso utilizaremos map (aplica una función a cada elemento de un array) y parseInt (convierte de string a número). Esto lo podemos hacer de la siguiente manera:

```
const result = ["1", "2", "3"].map((item) => parseInt(item));
console.log(result);
```

Esto funciona correctamente, nos devuelve por consola:
[1, 2, 3]

Si sólo tenemos un parametro de entrada para la función parseInt (o al menos parece que sólo tiene uno) no es un poco lata escribir eso de (item) => ... en el map, ¿ No podríamos saltarnoslo? Si quisiera usar map sin expresión lamda para convertir un array de strings en enteros tal que así:

```
["1", "2", "3"].map(parseInt); // => [1, NaN, NaN]

// parseInt( number = currentValue, base = index)
// parseInt(1, 0) => 1
// parseInt(2, 1) => NaN
// parseInt(3, 2) => NaN
```

El resultado es [1, NaN, NaN]. Esto ocurre porque la función map está definida para recibir una función con la firma function(currentValue, index, array) y parseInt de JavaScript (que normalmente lo usamos con un único parámetro), tiene otro segundo parámetro que es la base parseInt(number, base).

```
// Decorador unario
const unary = fn => {
  return (...args) => fn(args[0]);
}
```

Un decorador unario convierte cualquier función en una función unaria, es decir, una función que sólo tiene un parámetro de entrada.

Haciendo uso del decorador unario se puede hacer lo que quería de esta manera:

```
["1", "2", "3"].map(unary(parseInt)); //=> [1, 2, 3]
```

No tenemos por que implementarnos nuestra función unary librería como Ramda o Lodash/fp ya lo tienen implementado por nosotros.

El tema de funciones de orden superior (high order functions - HOF), está muy relacionados con los componentes de Ordern superior de React (High Order Components - HOC): un HOC es una función que toma un component y devuelve un nuevo components. Un ejemplo de HoC muy extendido es AutoSizer (le pasamos a un component el ancho que tiene disponible para dibujarse. Un ejemplo de como se usa:

```
  public render() {
    return (
      <div>
        <h1>Students</h1>
```

```
      <AutoSizer disableHeight={true}>
        {({ width }) => <StudentTableComponent width={width}
studentList={this.props.studentList} />}
      </AutoSizer>
      <Link to={adminRouteEnums.default}>Go back to dashboard</Link>
    </div>
  );
 }
```

Esto tiene muchas aplicaciones, por ejemplo podríamos implementar un Hoc que nos informara de los permisos del usuario logado, si tener que arrastrarlo desde el componente raíz.

Uso de currying (o curryficación)

Currificar consiste en convertir una función de múltiples variables en una secuencia de funciones unarias

Esto hace que, si la función tiene N argumentos de entrada, nunca se ejecutará si no le proporcionamos todos los argumentos de entrada que pide, al contrario de lo que ocurre por defecto en JavaScript (como con parseInt, que podíamos ejecutarla con un sólo argumento a pesar de recibir dos).

La currificación nos permite reutilizar una función en diferentes sitios con diferentes configuraciones.

Suponte que quieres calcular la suma de dos números:

```
const suma = (a, b) => a + b;
```
De manera que:

```
suma(3, 5); //=> 8
suma(3)(5); //=> TypeError
```
Si currificamos la función nos quedaría así:

```
const suma = (a) => (b) => a + b;
```
Esto lo hemos hecho utilizando sintaxis de ES6 (más fácil de leer cuando te acostumbras a ella), la misma función con sintaxis de ES5 quedaría de la siguiente manera:

```
function suma(a) {
  return function(b) {
    return a + b;
  }
}
```
¿ Qué llamadas podemos hacer ahora?

```
suma(3)(5); //=> 8

const sumPending = suma(3); // (b) => a + b
sumPending(2); // 8
```