

Desarrollo Backend

Bienvenid@s

High Order Functions I

Clase 08

UNTREF

UNIVERSIDAD NACIONAL
DE TRES DE FEBRERO



Pon a grabar la clase



Temario

- Array de datos
 - Estructura de un array de elementos vs. objetos
 - Métodos comunes
- Introducción a **High Order Functions**
 - Qué son las funciones de orden superior
 - Ejemplo de implementación
- Métodos avanzados en Arrays
 - el método **.forEach()**
 - el método **.find()**
 - el método **.filter()**
 - el método **.some()**
- Combinando métodos

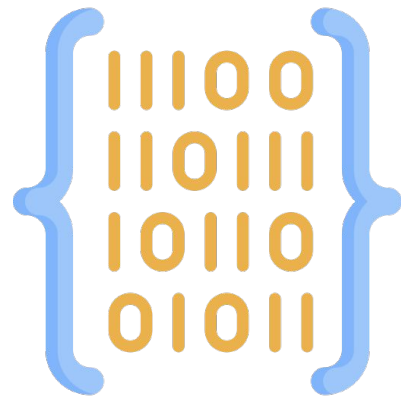


Array de Datos

Comenzaremos a trabajar de forma periódica con arrays.

Es importante que domines bien, tanto los arrays de elementos como los de objetos. Principalmente porque, estos últimos, son la estructura elegida por las aplicaciones backend para intercambiar información con otras aplicaciones.

Pasemos a refrescar conceptos



Array de Datos

Aquí, un array de elementos.

```
● ● ● Arrays
const paises = ['Argentina', 'Bolivia', 'Chile', 'Paraguay', 'Uruguay']
```

Por aquí, un array de objetos, también llamado array de objetos JSON.

```
● ● ● Arrays
const productos = [{id: 1, nombre: 'Notebook 14" FHD', importe: 115000},
  {id: 2, nombre: 'Tablet PAD 9.7"', importe: 195000},
  {id: 3, nombre: 'Macbook Air 13', importe: 745000},
  {id: 4, nombre: 'Tablet DROID 10.1"', importe: 165000},]
```

Array de Datos

Ambos utilizan los métodos convencionales de inserción y eliminación de elementos u objetos. Si por ejemplo necesitamos incrementar su contenido, le pasamos como parámetro un elemento, o un objeto a insertar.



```
países.push('Brasil')

productos.push({id: 5, nombre: 'Smartwatch 1.8"', importe: 22000})
```

Métodos comunes



Arrays

```
//último elemento del array  
países.pop()  
productos.pop()  
  
//primer elemento del array  
países.shift()  
productos.shift()
```

Para eliminar elementos, recurrimos al método **.pop()** que elimina el último, o al método **.shift()** que elimina el primer elemento del array.

Para eliminar un elemento intermedio, debemos primero identificar su posición (*índice*), y luego utilizar el método **.splice()** para removerlo.

UNTREF

UNIVERSIDAD NACIONAL
DE TRES DE FEBRERO

Métodos comunes



Arrays

```
//ubicar el índice de un elemento
const idx = paises.findIndex('Brasil')
if (idx > -1) {
  paises.splice(idx, 1)
}

//ubicar el índice en array de objetos
const idx = productos.findIndex(¿😬?)
productos.shift(idx, 1)
```

Trabajar la quita de elementos específicos es fácil cuando trabajamos con un array de elementos pero, al trabajar con un array de objetos, la cosa es algo diferente.

Para subsanar esto último, existen métodos que parten de lo que conocemos como funciones de orden superior.

UNTREF

UNIVERSIDAD NACIONAL
DE TRES DE FEBRERO

Introducción a High Order Functions

Qué son las funciones de orden superior

Las **High Order Functions**, también conocidas como **funciones de orden superior**, son funciones en JavaScript que aceptan una o más funciones como argumentos y/o devuelven una función como resultado.

Su principal tarea es operar fácilmente sobre otras funciones, tratarlas como datos y/o transformarlas.



Qué son las funciones de orden superior

Esto significa que, las High Order Functions, permiten que el código sea más modular y reutilizable, ya que las funciones pueden ser combinadas y anidadas para producir un comportamiento más complejo.



UNTREF

UNIVERSIDAD NACIONAL
DE TRES DE FEBRERO

Qué son las funciones de orden superior

Antes de entrar a las funciones de orden superior que integra el lenguaje JavaScript, veamos cómo podemos crear una función de orden superior a partir de nuestra propia necesidad.

A continuación, abordemos un pequeño ejemplo.



UNTREF

UNIVERSIDAD NACIONAL
DE TRES DE FEBRERO

Ejemplo de implementación

```
High Order Functions

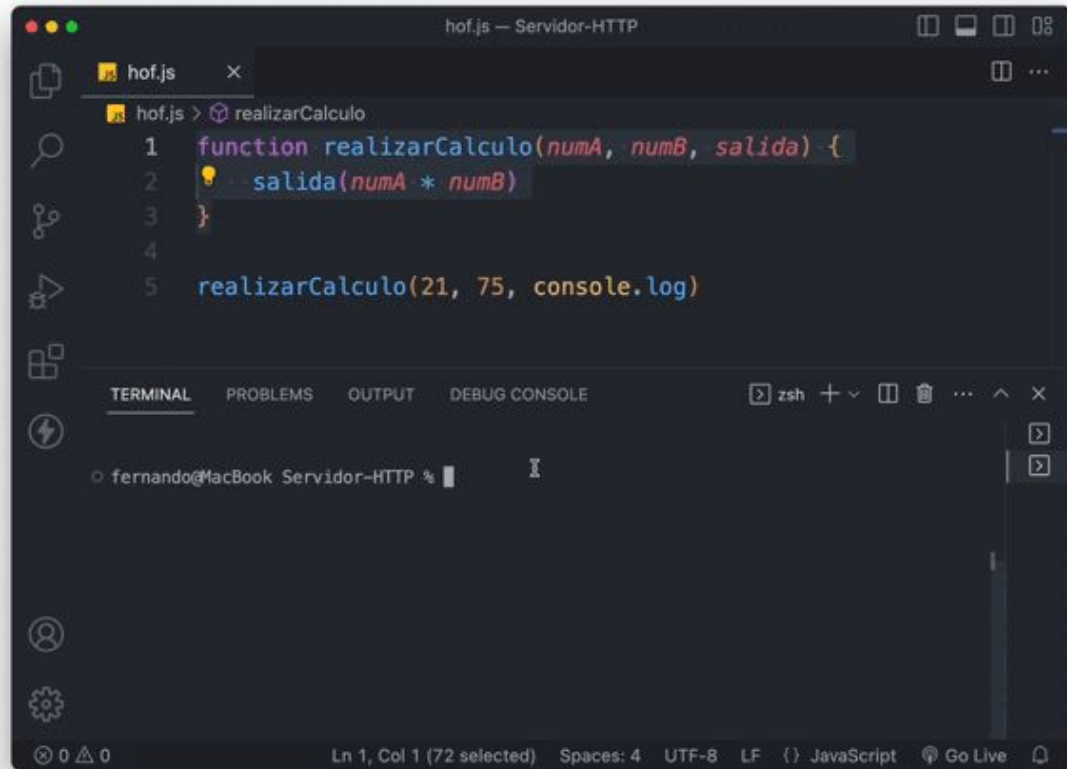
function realizarCalculo(numA, numB, salida) {
  salida(numA * numB)
}

realizarCalculo(21, 75, console.log)
```

Creamos una función llamada **realizarCalculo()**. La misma recibe tres parámetros, dos numéricos y un tercer parámetro, denominada **salida**.

Luego, llamamos a la función para ver su resultado.

Ejemplo de implementación



The screenshot shows a code editor window titled 'hof.js - Servidor-HTTP'. The editor contains the following JavaScript code:

```
1 function realizarCalculo(numA, numB, salida) {  
2   salida(numA * numB)  
3 }  
4  
5 realizarCalculo(21, 75, console.log)
```

The code is written in a dark-themed editor. The function `realizarCalculo` takes three parameters: `numA`, `numB`, and `salida`. It calls `salida` with the result of `numA * numB`. The function is then called with the arguments `21`, `75`, and `console.log`.

Below the code editor, there is a terminal window with the prompt `fernando@MacBook: Servidor-HTTP %`.

Los primeros parámetros, que son de entrada, reciben números, y el tercer parámetro, **salida**, recibe un objeto.

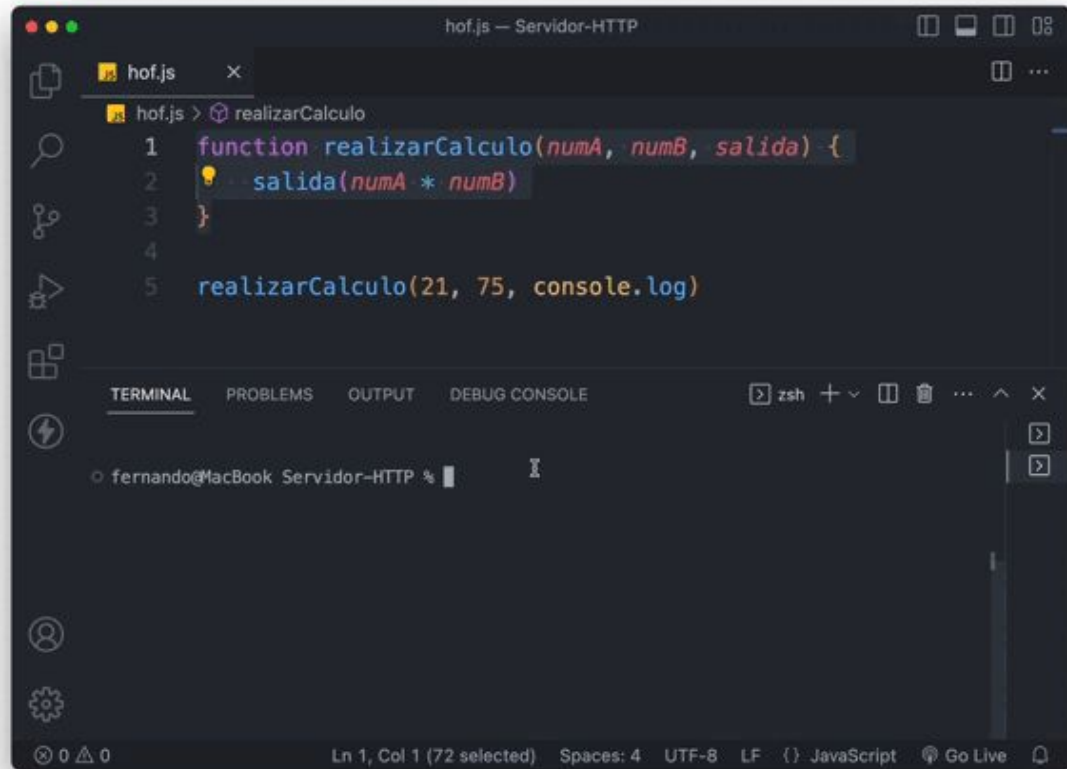
En este caso, es el objeto **console** y su método **.log()**.

Internamente, en la función, multiplicamos los números recibidos y los mostramos como resultado utilizando la función u objeto de salida.

UNTREF

UNIVERSIDAD NACIONAL
DE TRES DE FEBRERO

Ejemplo de implementación



The screenshot shows a code editor window titled 'hof.js - Servidor-HTTP'. The editor contains a JavaScript file 'hof.js' with the following code:

```
1 function realizarCalculo(numA, numB, salida) {  
2   salida(numA * numB)  
3 }  
4  
5 realizarCalculo(21, 75, console.log)
```

The editor interface includes a sidebar on the left with icons for Explorer, Search, Source Control, Run and Debug, Extensions, and Settings. At the bottom, there is a 'TERMINAL' tab showing the prompt 'fernando@MacBook Servidor-HTTP %'. The status bar at the bottom indicates 'Ln 1, Col 1 (72 selected)', 'Spaces: 4', 'UTF-8', 'LF', and 'JavaScript'.

En definitiva, lo que logramos con la función **realizarCalculo()** es poder llevar adelante una tarea específica, tomando distancia o abstrayéndonos de qué ocurre dentro de esta función en particular.

Y la lógica de una función de orden superior es justamente lograr realizar una tarea, y que esta se resuelva correctamente sin que tengamos que conocer cómo se resuelve.

Métodos avanzados en Arrays

Métodos avanzados en Arrays

Veamos entonces, cómo los arrays, tanto de elementos como también de objetos, incluyen una serie de métodos, que básicamente son funciones de orden superior.

Estos reciben una función como parámetro, y nos retornan en casi todos los casos, un resultado ya procesado sobre los datos del array en cuestión.



Métodos avanzados en Arrays

En una primera instancia, nos concentraremos en cuatro métodos específicos.

Estos son:

- `forEach()`
- `find()`
- `filter()`
- `some()`

Cada una de estas funciones, o métodos, acepta una o más funciones como argumentos y/o devuelve una función como resultado, permitiendo una mayor flexibilidad y modularidad en el código.



Métodos avanzados en Arrays

Método	Descripción
forEach()	Se utiliza para iterar sobre cada elemento de un array y ejecutar una función proporcionada para cada uno de ellos.
find()	Se utiliza para buscar un elemento en un array que cumpla con una determinada condición.
filter()	Se utiliza para filtrar los elementos de un array que cumplan con una determinada condición.
some()	Se utiliza para comprobar si al menos un elemento de un array cumple con una determinada condición.

Métodos avanzados en Arrays

Profundicemos cada uno de ellos para ver cómo usarlos en casos aplicados.

Trabajaremos sobre el siguiente modelo de array.

```
JS hof.js x
JS hof.js > ...
7  const productos = [{id: 1, nombre: 'Notebook 14" FHD', importe: 115000, categoria: 'Portátil'},
8                             {id: 2, nombre: 'Tablet PAD 9.7"', importe: 195000, categoria: 'Tablet'},
9                             {id: 3, nombre: 'Macbook Air 13', importe: 745000, categoria: 'Portátil'},
10                            {id: 4, nombre: 'Tablet DROID 10.1"', importe: 165000, categoria: 'Tablet'},
11                            {id: 5, nombre: 'Smartwatch 1.8" black', importe: 22500, categoria: 'Relojes'},
12                            {id: 6, nombre: 'Smartwatch 2" red', importe: 24200, categoria: 'Relojes'}]
13
14
```

forEach()

forEach()



High Order Functions

```
productos.forEach(producto => {  
  console.table(producto);  
})
```

El método **forEach()** itera sobre el array en cuestión, recorriendo el mismo desde el principio hasta el fin.

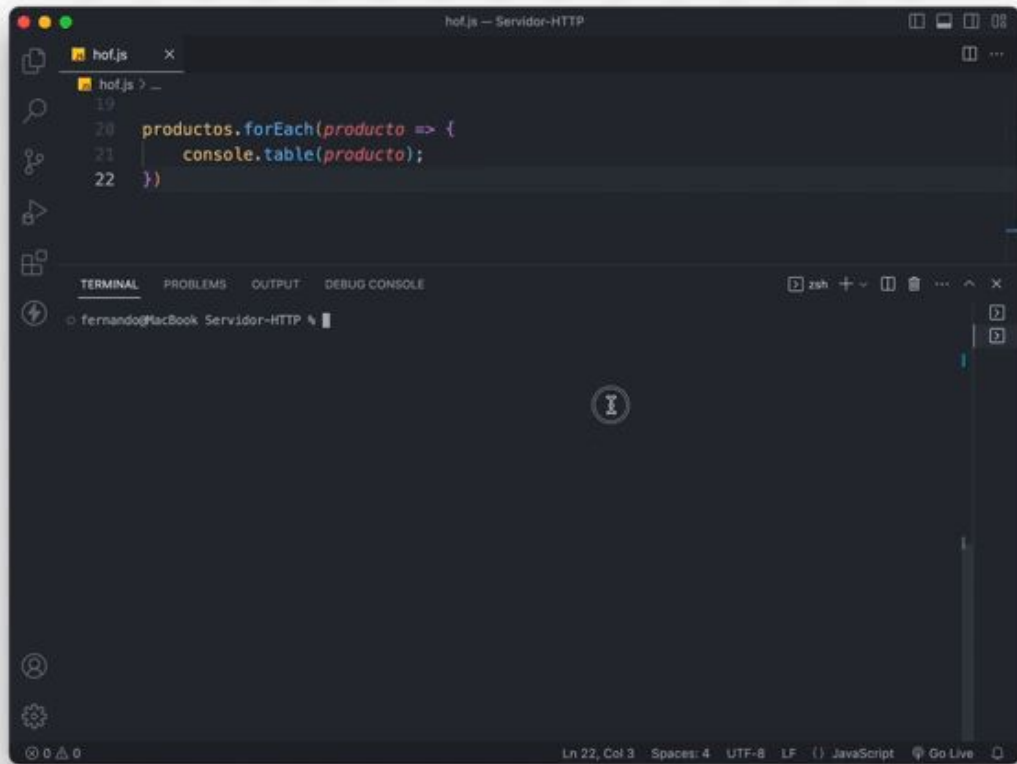
El método recibe un parámetro, denominado **predicado** el cual, por cada iteración, copia el elemento en el cual se está iterando.

Dentro de esta iteración podemos definir que se ejecute el código que consideremos necesario.

UNTREF

UNIVERSIDAD NACIONAL
DE TRES DE FEBRERO

forEach()



```
19
20 productos.forEach(producto => {
21   console.table(producto);
22 })
```

The screenshot shows a code editor window titled 'hof.js - Servidor-HTTP'. The editor contains a JavaScript file named 'hof.js' with the following code:
19
20 productos.forEach(producto => {
21 console.table(producto);
22 })
Below the code editor, there is a terminal window showing the command prompt 'fernando@MacBook Servidor-HTTP'.

En este ejemplo, ejecutamos **console.table()** para ver en formato tabla, cada uno de los elementos del array iterados.

Si bien el ciclo **for convencional**, o el ciclo **for...of** nos permiten realizar la misma tarea, **forEach()** define en el **predicado**, el elemento en el cual se está iterando.

Además está optimizado lo cual lo hace mucho más veloz si debemos iterar decenas o centenas de elementos de un array.

forEach()

Funciona por igual con un array de elementos. Puede que sea más apropiado su uso dentro de una aplicación frontend que en un backend, pero también lo tenemos disponible por si podemos sacar provecho de éste.



```
High Order Functions

const paises = ['Argentina', 'Brasil', 'Chile', 'Uruguay'];

paises.forEach(pais => {
  console.log(pais);
})
```


find()

find()

El método **find()** itera sobre el array en búsqueda de un elemento u objeto en particular.

En el **predicado** que recibe como parámetro, se asienta cada uno de los elementos u objetos que vamos iterando. Allí podemos aplicar una comparación con el elemento, o con la propiedad del objeto, en búsqueda de encontrar el primer elemento del array coincidente.

High Order Functions

```
const resultado = productos.find(producto => producto.id = 5)

if (resultado !== 'undefined') {
  console.table(resultado)
}
```

UNTREF

UNIVERSIDAD NACIONAL
DE TRES DE FEBRERO

find()

```
High Order Functions

const resultado = productos.find(producto => producto.id = 5)

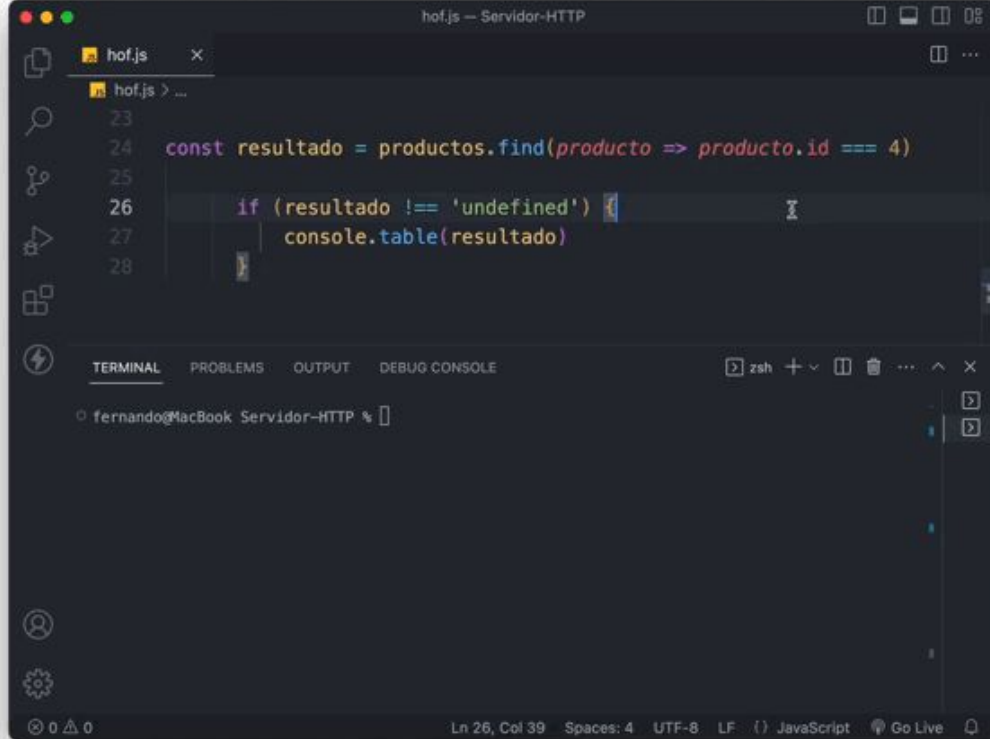
if (resultado !== 'undefined') {
  console.table(resultado)
}
```

Este método retorna un resultado al momento de encontrar la primera coincidencia, y dejar de iterar el array. Por ello, debemos definir una constante en la cual se asienta lo que el método `find()` nos retorna como resultado.

Ante la primera coincidencia que se encuentra, el método **find()** devuelve el objeto o elemento como resultado (*según el tipo de array*).

Si no encuentra coincidencia, devuelve **undefined**.

find()



```
hof.js — Servidor-HTTP
hof.js > ...
23
24 const resultado = productos.find(producto => producto.id === 4)
25
26 if (resultado !== 'undefined')
27   console.table(resultado)
28
```

fernando@MacBook: Servidor-HTTP %

Aquí podemos ver un ejemplo de los dos escenarios. Cuando el resultado es válido, porque el método **.find()** encontró un elemento coincidente, este elemento u objeto, es asignado a la constante **resultado**.

Caso contrario, la constante recibe como asignación el valor *'undefined'*.

find()

```
High Order Functions

const resultado = productos.find(producto => producto.id = 150)

if (resultado !== 'undefined') {
  console.table(resultado)
}
```

Es clave que, luego de realizar una búsqueda con el método `find()`, controlemos qué resultado nos devuelve la misma. De esta forma tendremos un control sobre el mismo.

Si el resultado es válido, lo utilizamos en alguna otra operación. Si no, podemos mostrar algún mensaje de error “*amigable para el usuario*”.

find()

```
High Order Functions

const paises = ['Argentina', 'Brasil', 'Chile', 'Uruguay'];

const resultado = paises.find(pais => pais === 'Argentina');

if (resultado !== 'undefined') {
  console.log(resultado);
}
```

Aquí te compartimos un ejemplo, trabajando sobre un array de elementos.

filter()

filter()

```
High Order Functions

const resultado = productos.filter(producto => producto.categoria === 'Tablet')

if (resultado !== []) {
  console.table(resultado)
}
```

El método **filter()** se comporta de forma similar al método **find()**, con la diferencia de que **filter()** aplica un filtro sobre todos los elementos coincidentes del array.

Por ende, recorre el array de principio a fin, capturando todos aquellos elementos u objetos donde encuentra coincidencia. Finalmente, los devuelve como resultado en un nuevo array.

filter()

```
High Order Functions

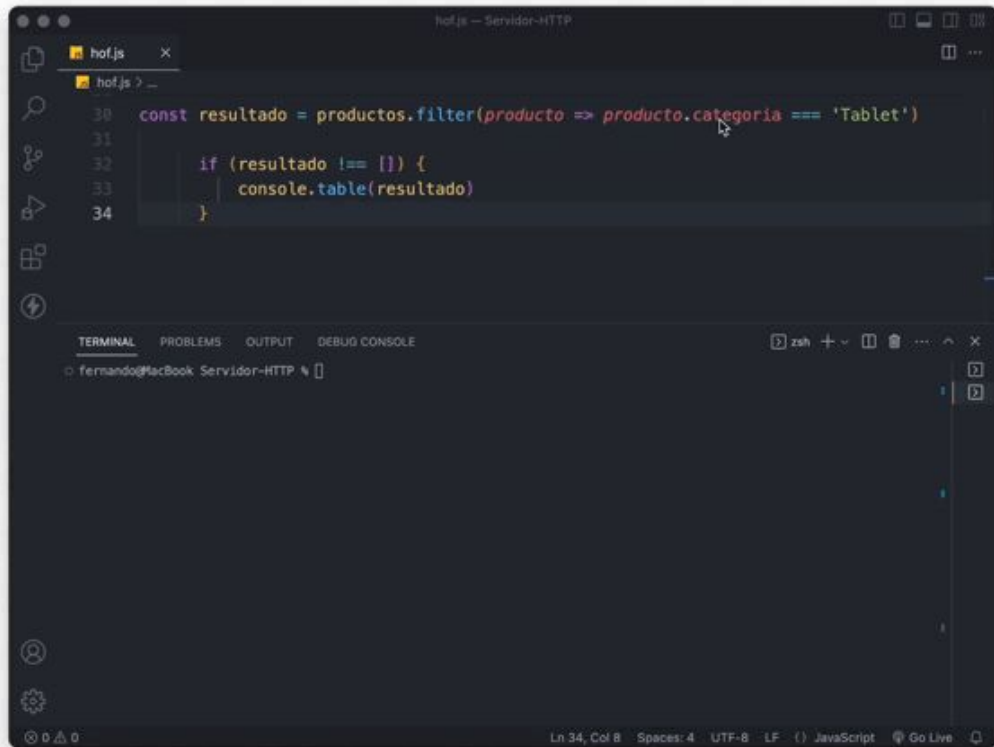
const resultado = productos.filter(producto => producto.categoria === 'Tablet')

if (resultado !== []) {
  console.table(resultado)
}
```

En el caso que no encuentra ninguna coincidencia, el método **filter()** devuelve como resultado un array vacío.

Debemos también tener la precaución de evaluar qué resultado nos devuelve este método, previo a trabajar con el array resultante. Si el array está vacío, podremos mostrar un mensaje de error alternativo, *“amigable para el usuario”*.

filter()



```
30 const resultado = productos.filter(producto => producto.categoria === 'Tablet')
31
32 if (resultado !== []) {
33   console.table(resultado)
34 }
```

TERMINAL PROBLEMS OUTPUT DEBUG CONSOLE

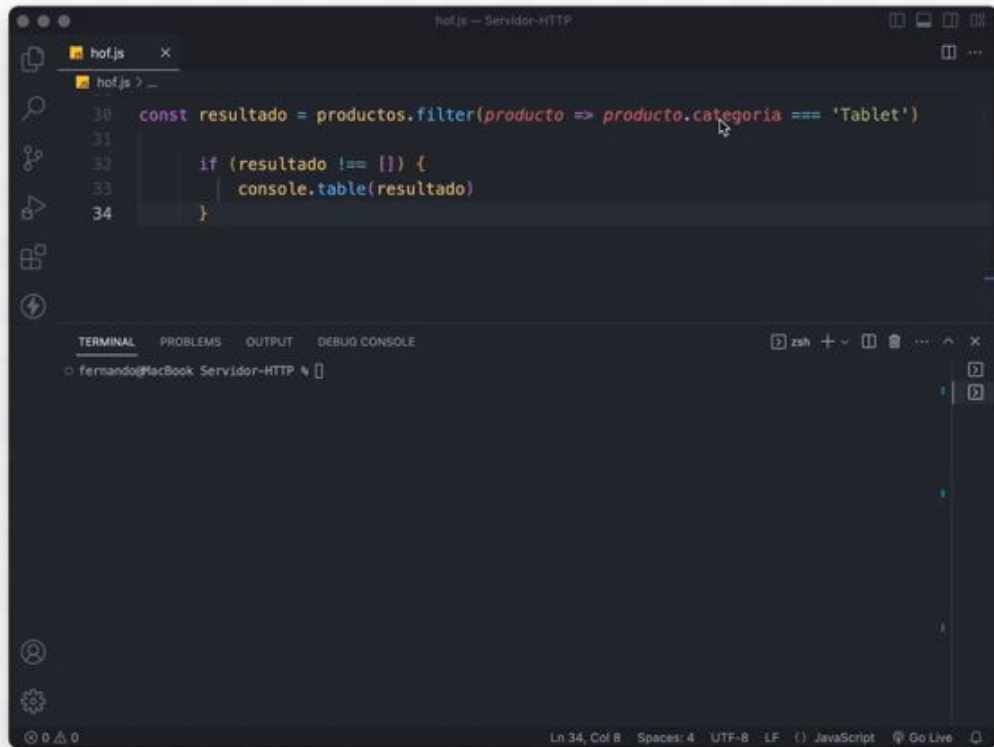
fernando@MacBook: Servidor-HTTP %

Ln 34, Col 8 Spaces: 4 UTF-8 LF JavaScript Go Live

Cuando validamos el array resultante del método `.find()`, podemos optar por preguntar si **array !== []** (*distinto de vacío*) o si **array.length > 0** (*al menos 1 elemento*).

Son dos posibles opciones a utilizar como condicional, para luego comenzar a trabajar con el array resultante.

filter()



```
30 const resultado = productos.filter(producto => producto.categoria === 'Tablet')
31
32 if (resultado !== []) {
33   console.table(resultado)
34 }
```

Podemos utilizar cualquiera de las propiedades de un array de objetos como parámetro para aplicar un filtro.

.filter() es más útil cuando debemos trabajar con un filtro en base a una cadena de texto (*nombre de un producto, categoría, etcétera*). Pero, en estos casos, suele surgir otra duda:

¿Y si quiero filtrar por una palabra específica de un nombre de producto compuesto? 🤖

filter()

Aquí es donde comienza la magia de JavaScript. Si el elemento o la propiedad es un tipo de dato string, entonces el mismo dispone del método `.includes('valor')`. Este método nos permite definir un texto o parámetro parcial, para así poder aplicar un filtro que no tenga exactitud.

Mira el ejemplo a continuación:

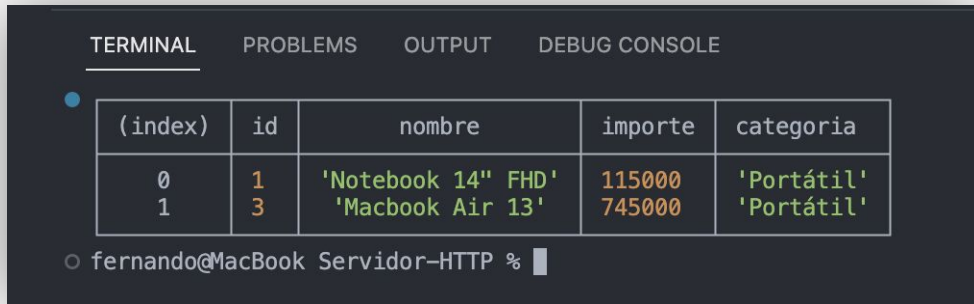
```
High Order Functions

const resultado = productos.filter(producto => producto.nombre.includes('book'))

if (resultado !== []) {
  console.table(resultado)
}
```

filter()

En nuestro array modelo disponemos de dos productos los cuales tienen la palabra “*Macbook*” y “*Notebook*”, respectivamente. El método **.filter()** recibe en el **predicado** parte de una palabra, y termina aplicando el filtro sobre ese valor.



The screenshot shows a terminal window with a table of products. The table has five columns: (index), id, nombre, importe, and categoria. The first row has index 0, id 1, nombre 'Notebook 14" FHD', importe 115000, and categoria 'Portátil'. The second row has index 1, id 3, nombre 'Macbook Air 13', importe 745000, and categoria 'Portátil'.

(index)	id	nombre	importe	categoria
0	1	'Notebook 14" FHD'	115000	'Portátil'
1	3	'Macbook Air 13'	745000	'Portátil'

fernando@MacBook Servidor-HTTP %

¿Y si el término que usamos como parámetro de búsqueda viene en mayúsculas? 😞

PD: ¡QUIERO RETRUCO!

filter()

También podemos sortear este obstáculo con JavaScript. Para ello, lo ideal sería guardar en una variable o constante el parámetro en cuestión, aplicando el método **.toLowerCase()** sobre esta para forzar su valor a minúsculas.

Luego, aprovechamos el encadenamiento de métodos que tiene JavaScript para hacer lo mismo con la propiedad sobre la cual hacemos la comparación.

```
High Order Functions

const parametro = 'BOOK'
const resultado = productos.filter(producto => {
  return producto.nombre.toLowerCase().includes(parametro.toLowerCase())
})

if (resultado !== []) {
  console.table(resultado)
}
```

filter()

Más allá de implementar este truco en los filtros de datos, también podemos hacerlo totalmente aplicable a situaciones específicas donde utilizamos el método **find()**.



High Order Functions

```
const parametro = 'taBLeT'

const resultado = productos.find(producto => {
  return producto.categoria.toLowerCase() === parametro.toLowerCase()
})
```

UNTREF

UNIVERSIDAD NACIONAL
DE TRES DE FEBRERO

filter()

Y, tanto **filter()** como **find()**, también nos permiten definir en una búsqueda o filtro de resultados, la combinación de dos o más parámetros específicos, tal como vemos en el siguiente ejemplo.

```
High Order Functions

const parametro = 'taBLeT'

const resultado = productos.find(producto => {
  return producto.categoria.toLowerCase() === parametro.toLowerCase()
  && producto.nombre.includes("DROID")
})
```


filter()

Este tipo de prácticas es muy recomendada para sortear cualquier obstáculo referente a cómo se cargaron datos o cómo se recibe un valor el cual luego debe usarse para buscar o filtrar información.

Si bien estamos del lado del backend y el parámetro de búsqueda debe provenir de una aplicación web frontend, la cual también debería normalizar el dato a enviar a un backend, no está mal aplicar múltiples controles, siempre.

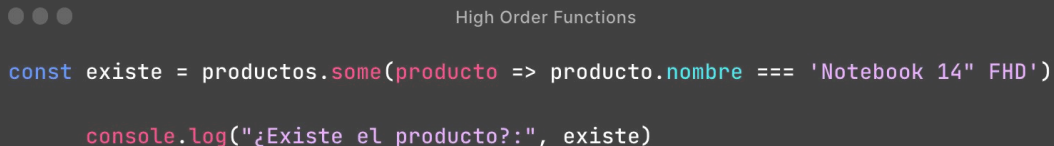
```
High Order Functions

const parametro = 'PORtáTil'
const resultado = productos.filter(producto => {
  return producto.categoria.toLowerCase().includes(parametro.toLowerCase())
})
```

some()

some()

El método **some()** nos permite comprobar si, al menos, un elemento de un array cumple con una determinada condición. Devuelve un valor booleano **true** si al menos un elemento cumple con la condición, y **false** si ninguno de los elementos la cumple.

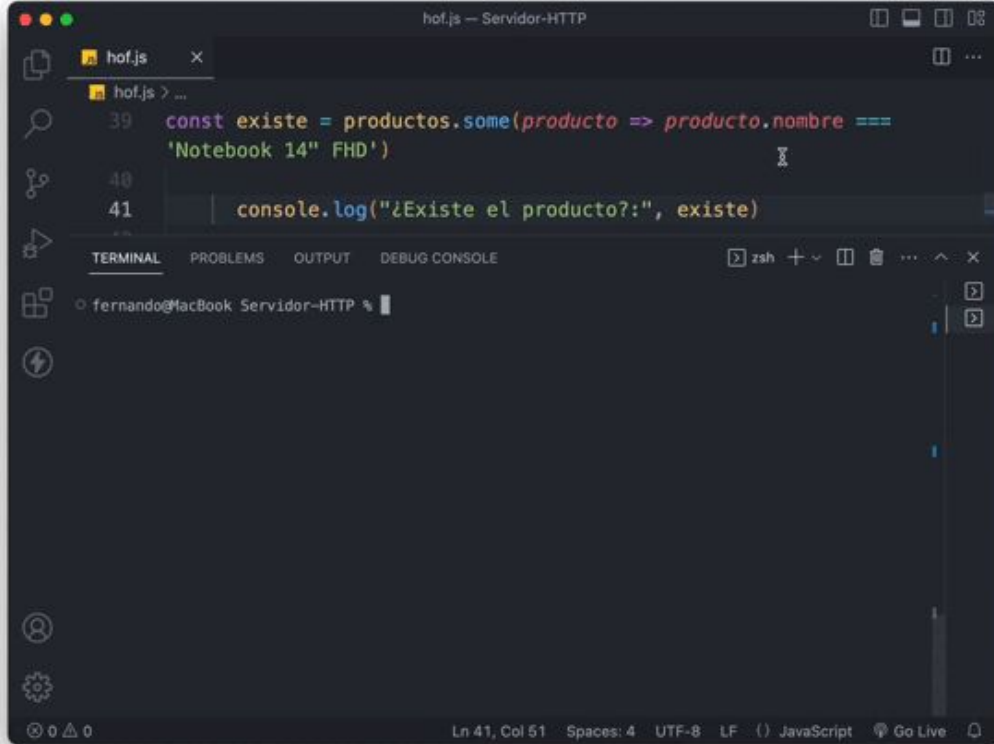


```
High Order Functions

const existe = productos.some(producto => producto.nombre === 'Notebook 14" FHD')

console.log("¿Existe el producto?", existe)
```

some()



```
hof.js — Servidor-HTTP
hof.js
39 const existe = productos.some(producto => producto.nombre ===
   'Notebook 14" FHD')
40
41 console.log("¿Existe el producto?:", existe)
```

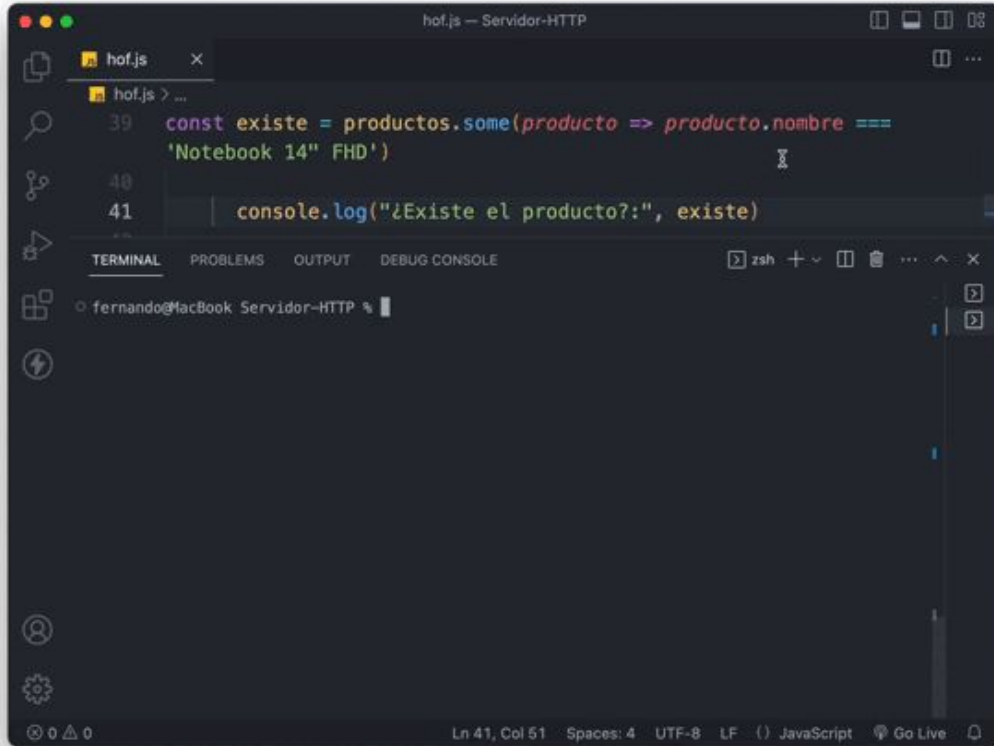
fernando@MacBook Servidor-HTTP %

Ln 41, Col 51 Spaces: 4 UTF-8 LF () JavaScript Go Live

El método **some()** detiene la iteración tan pronto como encuentra el primer elemento que cumple con la condición.

Por lo tanto, si hay varios elementos en el array que cumplen con la condición, el método **some()** sólo devuelve **true** para el primer elemento que cumple con la condición y no comprueba los demás elementos.

some()



```
hof.js — Servidor-HTTP
hof.js > ...
39 const existe = productos.some(producto => producto.nombre ===
   'Notebook 14" FHD')
40
41 console.log("¿Existe el producto?:", existe)
```

fernando@MacBook Servidor-HTTP %

Ln 41, Col 51 Spaces: 4 UTF-8 LF () JavaScript Go Live

Su aplicación es efectiva para casos como, por ejemplo, validar si un producto existe previo a darlo de alta en un sistema.

Así nos aseguramos de no generar duplicados en la información que se carga.

En aplicaciones de backend, es fundamental que apliquen este tipo de controles sobre información que deba almacenarse.

Funciones de orden superior



Como podemos ver, las funciones de orden superior son herramientas muy útiles para trabajar con un conjunto de datos (Array), y tomar mejores decisiones de acuerdo a los resultados.

A su vez, son de gran ayuda para no tener que estar peticionando información de manera constante contra una base de datos, generando en estas situaciones una congestión en la misma, en momentos donde el tráfico de red es alto.

Leer un archivo

Aún no ha terminado nuestro recorrido por las diferentes funciones de orden superior. Todavía nos queda una clase más para seguir viendo diferentes opciones que propone el lenguaje JavaScript, para aquellos momentos donde necesitamos trabajar de la forma más efectiva, manipulando volúmenes de datos.

Realicemos a continuación una práctica con todo lo aprendido hasta aquí.



Espacio de trabajo

Espacio de trabajo

Tomar el ejemplo asincrónico de la explicación elaborada, y convertirlo a Promesas.

Tiempo estimado: 20 minutos. 

```
const questions = [ 'dudas', 'consultas', '🤔' ]
```



```
> node gracias.js
```

UNTREF

UNIVERSIDAD NACIONAL
DE TRES DE FEBRERO