

JavaScript Avanzado

WebStorage - Operadores avanzados - Módulos JS



Agenda de hoy

- **Módulos JS**

- export e import
- export default
- type module

- **Operadores avanzados**

- operador ternario
- operador lógico AND
- operador lógico OR
- operador de acceso condicional
- desestructuración de objetos y elementos
- alias en desestructuración
- Object.groupBy



Módulos JS



Módulos JS

Hasta ahora, la forma predeterminada de declarar archivos JS con la que hemos estado trabajando es referenciar los mismos en el apartado **<head>** de los documentos HTML.

Si bien no está mal esta forma de trabajar, desde la llegada de **ES6** existe otra manera de simplificar este tipo de declaraciones: *integrando módulos JS*.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="UTF-8">
  <title>ABM Productos</title>
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <link rel="stylesheet" href="css/styles.css">
  <script src="js/productos.js"></script>
  <script src="js/claseProducto.js"></script>
  <script defer src="js/main.js"></script>
</head>
<body>
  <h1>ABM Productos</h1>
  ...
```

Módulos JS

La capacidad de declarar archivos JavaScript como módulos se introdujo en el estándar ECMAScript 6, en 2015.

Esta especificación proporciona una forma estándar y más moderna de estructurar y modularizar código JavaScript en aplicaciones frontend.

```
...
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="UTF-8">
  ...
  <script type="module" src="js/main.js"></script>
  ...
</head>
```

Módulos JS

Dentro de las claras ventajas que encontramos implementando el estos, destacamos:

- Encapsulamiento y scope local
- Reutilización y organización de código
- Carga de contenido bajo demanda
- Gestionar dependencias eficazmente
- Utilizar Strict Mode por defecto
- Importar de forma asincrónica
- Legibilidad y mantenimiento del código
- Diferir la carga del archivo JS por defecto

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="UTF-8">
  ...
  <script type="module" src="js/main.js"></script>
  ...
</head>
```

Módulos JS

Ventaja	Descripción
Encapsulamiento y scope local	Los módulos de JavaScript tienen su propio ámbito local. Esto significa que las variables y funciones definidas en un módulo no se filtran al ámbito global, evitando colisiones de nombres y problemas de superposición global.
Reutilización de código	Podemos exportar funciones, objetos o variables desde un módulo y luego importarlos en otros módulos, fomentando la reutilización de código, gracias a que usamos las mismas funciones y objetos en múltiples partes de la aplicación.
Organización de código	Los módulos nos permiten organizar el código de manera más lógica y modular. Podemos dividir la aplicación en partes más pequeñas y manejables, facilitando el mantenimiento y la colaboración en proyectos más grandes.
Cargar contenido bajo demanda	Los módulos se cargan utilizando (<i>lazy loading</i>), o sólo cuando se importan explícitamente en otros módulos. Esto mejora significativamente el rendimiento de la aplicación, ya que no se cargan todos los scripts al principio.
Gestionar dependencias eficazmente	Podemos especificar las dependencias de un módulo en la parte superior de nuestros archivos JS, facilitando la comprensión de las relaciones entre los diferentes componentes de la aplicación web.
Utilizar <i>Strict Mode</i> por defecto	Los módulos JavaScript se ejecutan en modo estricto (" <i>use strict</i> ") de forma predeterminada, ayudando a evitar errores comunes y mejorando la calidad del código.
Diferir la carga del archivo JS por defecto	El atributo defer utilizado para referenciar el archivo JS en el <head> de cada documento, pasa a ser opcional, ya que la carga diferida de JS es la opción predeterminada para los archivos/módulos JS.



Módulos JS

Ventaja	Descripción
importar de forma asincrónica	Podemos importar asincrónicamente con import() y cargar módulos bajo demanda; algo útil para cargar dinámicamente partes de la aplicación solo cuando es necesario.
Legibilidad y mantenimiento del código	Los módulos ayudan a dividir el código en componentes más pequeños y enfocados, lo que facilita la lectura, comprensión y mantenimiento del código.

Cuando utilizamos módulos JS en aplicaciones frontend, ganamos en modularidad, encapsulamiento, reutilización de código y organización, lo que conduce a un código más limpio, mantenible y eficiente. Esto es valioso en proyectos de porte mediano a grande y, sobre todo en proyectos complejos, con gestión de código crítica.

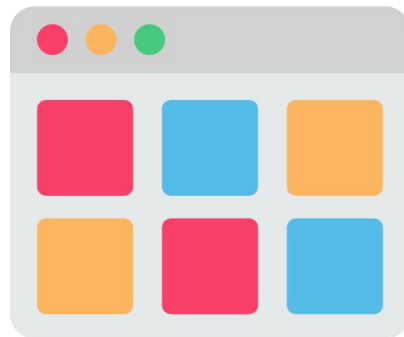
Además, es el principio de programación de Frameworks y Librerías JS.



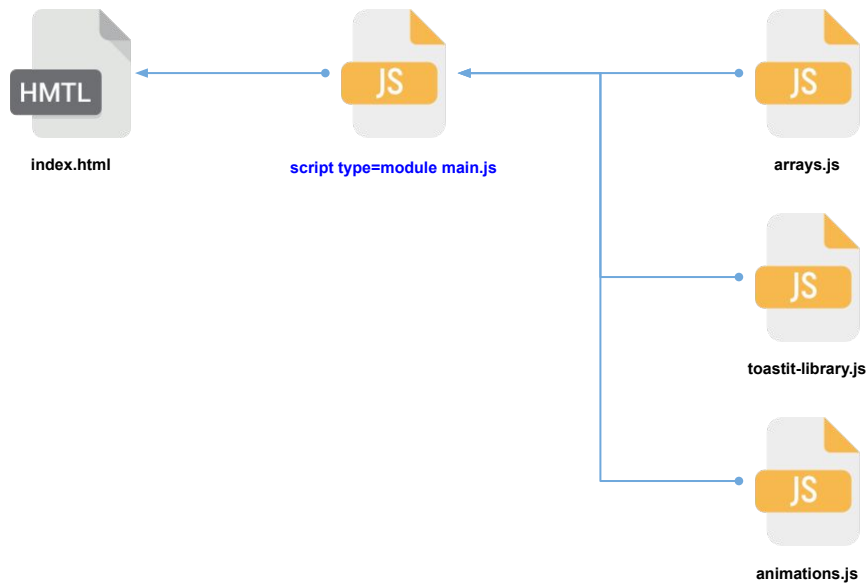
Módulos JS

Con la introducción de módulos JS en EcmaScript 6, se incorporaron las palabras clave **import** y **export**, las cuales permiten importar y exportar funciones, objetos y variables entre módulos.

Esto revolucionó la forma de organizar y gestionar el código JS en aplicaciones web.



Módulos JS



Nuestro documento HTML solo tendrá un único archivo JS referenciado como módulo.

Luego, si este utiliza una o más funciones, variables, constantes, arrays, objetos o cualquier otra referencia de uno o más archivos JS adicionales, sólo debemos importar estos objetos o elementos específicos a nuestro archivo JS definido como un módulo.

Así lograremos beneficios como, evitar cargar toda una librería JS de terceros en la memoria de nuestra aplicación web, cuando sólo necesitamos utilizar una o dos funciones de ésta.

Módulos JS

Si tenemos un archivo llamado **animaciones.js** con funciones, arrays y constantes que se exportan para ser utilizadas en otros archivos JS, nuestro código funcional será similar al de este ejemplo.

```
animaciones.js

function aparecer(milliseconds) {
  setTimeout(() => console.log('Aparecer en', milliseconds, 'ms.'), milliseconds)
}

const clasesCSS = ['clase1', 'clase2', 'clase3']

const apiKey = 'mi_clave_secreta'

export { aparecer, clasesCSS, apiKey }
```

```
main.js

import { aparecer, clasesCSS, apiKey } from './animaciones.js'

aparecer(2000)
console.log(clasesCSS)
console.log(apiKey)
```

Luego, desde nuestro archivo JS declarado como módulo, exportamos solo el contenido que necesitamos utilizar. Este podrá ser invocado luego, tal como si lo hubiésemos creado de forma local en el archivo JS principal.

Módulos JS

Dentro de las ventajas que encontramos con los módulos JS, podemos destacar:

- **carga parcial de contenido**

(evitamos cargar funciones, variables, constantes, objetos, etc, que no sean utilizados por la aplicación)

- **comportamiento de ejecución diferida del código JS**

(JS se carga luego de todo el HTML, sin necesidad de utilizar el atributo defer en el módulo JS principal)

- **variables y constantes con scope local**

- **entender cómo se crearán aplicaciones JS basadas en**

frameworks JavaScript



JS: Operadores modernos/avanzados



Operadores modernos/avanzados

Desde el lanzamiento de **ES6** (2015), JavaScript comenzó a incluir un montón de operadores modernos en el lenguaje.

Estos se basan en el paradigma que se denomina, en muchos lenguajes, como “**Syntactic Sugar**”. Este nos permite escribir una sintaxis diferente y simplificada, que permite hacer lo mismo que la sintaxis convencional, comúnmente más extensa.



Operadores modernos/avanzados

Un ejemplo de ello es lo que vemos en el ciclo **for convencional**, y el ciclo **for...of**.

Este último, nos abstrae de la complejidad y control que aporta el ciclo for convencional, además que nos ayuda a escribir algo menos de código, y el código escrito es más claro de leer.

```
const array = ['JS', 'ES', 'CSS', 'HTML']

for (let i = 0; i < array.length; i++) {
  console.log(array[i]);
}
```



```
const array = ['JS', 'ES', 'CSS', 'HTML']

for (let el of array) {
  console.log(el);
}
```


Operadores modernos/avanzados

Incluso el parámetro `i++` que vemos
en un ciclo for convencional, es una
abreviatura de lo que sería:

```
i = i + 1
```

sintaxis que pasó también por una
simplificación previa:

```
i += 1
```




```
const array = ['JS', 'ES', 'CSS', 'HTML']  
  
for (let i = 0; i < array.length; i++) {  
  console.log(array[i]);  
}
```


Operador ternario

Operador ternario

El operador ternario en JavaScript nos brinda una forma concisa de expresar una estructura condicional más simplificada que un **if - else**.



```
condición ? expresiónSiVerdadero : expresiónSiFalso
```

The diagram shows a code editor window with a dark background. Inside, the ternary operator syntax is displayed: 'condición ? expresiónSiVerdadero : expresiónSiFalso'. Each part is underlined with a different color: 'condición' is yellow, 'expresiónSiVerdadero' is green, and 'expresiónSiFalso' is blue. Above the code, there are three small gray circles representing window controls.

- Se evalúa la **condición**
- Si es verdadera, se devuelve **expresiónSiVerdadero**
- Si es falsa, se devuelve **expresiónSiFalso**

Operador ternario

En este ejemplo, la condición (**edad** **>= 18**) se evalúa como verdadera, por lo que se retorna "*Eres mayor de edad*".

Si la edad fuera menor de 18, retorna el "*Eres menor de edad*".

```
let edad = 20
let mensaje = (edad >= 18) ? "Eres mayor de edad" : "Eres menor de edad"

console.log(mensaje); // Se imprime "Eres mayor de edad"
```

Operador ternario

Es útil cuando debemos tomar decisiones simples basadas en una condición y asignar un valor a una variable o expresión en función de esa condición, todo en una sola línea de código.

```
let edad = 20
let mensaje = (edad >= 18) ? "Eres mayor de edad" : "Eres menor de edad"

console.log(mensaje); // Se imprime "Eres mayor de edad"
```

Este mecanismo de acción es conocido como “*retorno implícito*”.



Operador lógico AND

Operador lógico AND

El operador lógico **&&** en JavaScript se utiliza para realizar operaciones lógicas de "y" (AND) en expresiones booleanas.

Como una comparación rápida, podemos ejemplificar que su uso puede aplicar en reemplazo de un **if** simple.

```
expresión1 && expresión2
```

Operador lógico AND

El operador **&&** utiliza una técnica llamada "*evaluación cortocircuito*" (**short-circuit evaluation**) lo que significa que, la segunda expresión (**expresión2**) solo se evaluará si la primera expresión (**expresión1**) es verdadera.

Si **expresión1** es falsa, JS no evaluará **expresión2** porque ya sabe que **&&** en su conjunto será falsa, independientemente de la evaluación de **expresión2**.

```
const esMayorDeEdad = true
const tienePermiso = true

const puedeIngresar = esMayorDeEdad && tienePermiso
```



Operador lógico AND

```
const esMayorDeEdad = true
const tienePermiso = true

const puedeIngresar = esMayorDeEdad && tienePermiso
```

En este ejemplo, la variable **puedeIngresar** será verdadera sólo si tanto **esMayorDeEdad** como **tienePermiso** son verdaderas.

Si cualquiera de las dos expresiones es falsa, **puedeIngresar** será falsa.

Operador lógico OR



Operador lógico OR

En JavaScript, el operador lógico **OR** se representa con `||` y se utiliza para evaluar expresiones lógicas o condiciones.

El operador **OR** devuelve **true** si al menos uno de los operandos es true.



```
expresion1 || expresion2
```

Operador lógico OR

El operador `||` se coloca entre dos expresiones que se desean comparar. La sintaxis básica es la siguiente:

```
expresion1 || expresion2
```

Evaluación perezosa (Short-Circuit):

JS usa una evaluación perezosa para el operador `||`, por lo cual si la primera expresión es **true**, no se evaluará la segunda expresión, ya que el resultado será **true** independientemente de la segunda expresión.



Operador lógico OR

Esto es útil para evitar errores en casos en los que la segunda expresión podría generar una excepción.

```
let resultado = true || false // resultado es true
```

El operador `||` funciona con cualquier tipo de operando, no solo valores booleanos. JS realizará una conversión automática a booleano en función de reglas de verdad (**truthy** y **falsy**).



Operador lógico OR

Esto es útil para evitar errores en casos en los que la segunda expresión podría generar una excepción.

Valor	Resultado
<code>0 "falsy"</code>	<code>"falsy"</code>
<code>' ' "falsy"</code>	<code>"falsy"</code>
<code>null "falsy"</code>	<code>"falsy"</code>
<code>undefined "falsy"</code>	<code>"falsy"</code>
<code>false "falsy"</code>	<code>"falsy"</code>
<code>NaN "falsy"</code>	<code>"falsy"</code>

Operator Nullish Coalescing



Operador Nullish Coalescing


El operador *nullish coalescing* (`??`) es un operador introducido en ES6 utilizado para proporcionar un valor de respaldo en caso de que una expresión sea **null** o **undefined**.



```
expresion1 ?? expresion2
```

Operador Nullish Coalescing

A diferencia del operador lógico OR el cual considera valores "*falsy*" a `false`, `0`, `' '`, etc., nullish coalescing solo devuelve el valor de respaldo si la expresión de la izquierda es estrictamente **null** o **undefined**.



```
expresion1 ?? expresion2
```


Operador Nullish Coalescing

El operador **??** verifica si la expresión de la izquierda (**expresion1**) es estrictamente igual a **null** o **undefined**. Si lo es, retorna la expresión de la derecha (**expresion2**), si no retorna la expresión de la izquierda.

```
let nombre = null
let nombrePredeterminado = "Invitado"
let nombreFinal = nombre ?? nombrePredeterminado

// nombreFinal es "Invitado", porque nombre es null
```

Operador de acceso condicional

Operador de acceso condicional

```
let valor = objeto.propiedad?.subpropiedad  
  
let otroValor = objeto2?.propiedadPrincipal
```

El operador de acceso condicional (`?.`) permite realizar una acción condicional si la propiedad o subpropiedad a la que se accede, existe.

Operador de acceso condicional

Si alguna de las propiedades a lo largo de la cadena de acceso tiene como valor: **null** o **undefined**, la expresión retorna **undefined**, evitando así tener que caer en un posible error en la lógica del código.

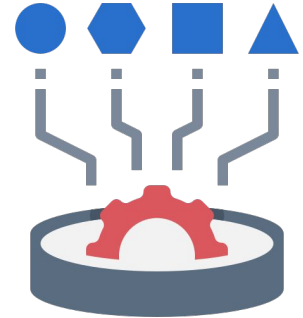
```
const persona = {  
  nombre: "Alice",  
  direccion: {  
    ciudad: "Nueva York",  
    codigoPostal: null  
  }  
}  
  
const codigoPostal = persona.direccion?.codigoPostal  
// codigoPostal es null, no se produce un error
```

Desestructuración



Desestructuración

La desestructuración es una característica que nos permite extraer valores de objetos y arreglos de manera más concisa y legible, pudiendo asignar estos valores a variables individuales.



Desestructuración

```
const persona = { nombre: "Joe", edad: 38, puesto: "CEO McMillian Sec." }  
const { nombre, edad } = persona  
  
console.log(nombre) // "Joe"  
console.log(edad)   // 38
```

La desestructuración de objetos nos permite extraer valores de propiedades de objetos y asignarlos a variables con el mismo nombre que las propiedades. Todo, en una sola línea de código.

Desestructuración

```
const persona = { nombre: "Ana" }  
const { nombre, edad = 25 } = persona  
  
console.log(nombre) // "Ana"  
console.log(edad)   // 25 (valor predeterminado)
```

También podemos asignar valores predeterminados a una propiedad, si esta no existe en el objeto.

Si existe, toma el valor del mismo, si no, utiliza el valor asignado de forma predeterminada.

Desestructuración

```
const persona = { nombre_de_pila: "Joe", edad_de_la_persona: 28 }

const { nombre_de_pila: nombre, edad_de_la_persona: edad } = persona

console.log(nombre)      // "Joe"
console.log(edad)        // 28
```

Incluso podemos utilizar alias al definir variables.

De esta manera, si el nombre de la propiedad a desestructurar es demasiado extenso o complejo, con un alias creamos la variable desestructurando la propiedad de forma más simple.



Desestructuración

```
const colores = ["rojo", "verde", "azul"]  
const [color1, color2, color3] = colores  
  
console.log(color1) // "rojo"  
console.log(color2) // "verde"  
console.log(color3) // "azul"
```

La desestructuración de arrays nos permite también extraer valores de un arreglo y asignarlos a variables en el orden en que aparecen en el primero.

Desestructuración

```
const [primero, , tercero] = colores  
  
console.log(primero) // "rojo"  
console.log(tercero) // "azul"
```

También puedes omitir elementos utilizando comas con un espacio en blanco entre valores.

Desestructuración

Y, por supuesto, es posible también desestructurar propiedades de un objeto directamente en los parámetros de una función, respetando el nombre correspondiente de cada uno de ellos.

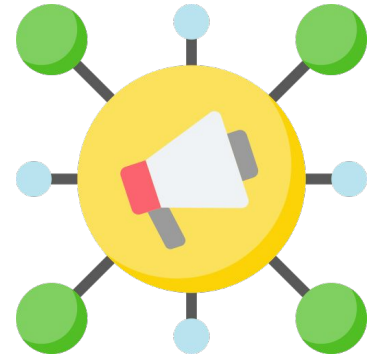
```
function imprimirDatos({ nombre, edad }) {  
  console.log(`Nombre: ${nombre}, Edad: ${edad}`)  
}  
  
const persona = { nombre: "Laura", edad: 35 }  
  
imprimirDatos(persona)  
// Se imprime en la consola: "Nombre: Laura, Edad: 35"
```

Spread Operator / Rest Parameters



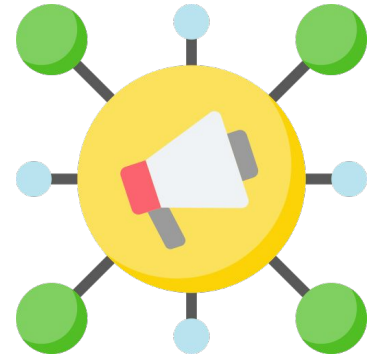
Spread Operator / Rest Parameters

El operador de expansión (Spread Operator, está representado por tres puntos consecutivos (...), y es una característica utilizada para descomponer o expandir elementos de arrays y objetos en lugares donde múltiples elementos o propiedades son esperados.



Spread Operator / Rest Parameters

En otras palabras, permite descomponer un objeto o un array en elementos individuales. Es muy útil en situaciones donde necesitamos combinar múltiples elementos en una nueva estructura de datos o pasar argumentos de manera más flexible.



Spread Operator / Rest Parameters

Aquí tenemos algunos ejemplos comparando el uso de **Spread**

Operator para concatenar contenido de un array dentro de otro array.

La última línea de código, representa una alternativa utilizada anteriormente para este propósito.

```
const originalArray = [1, 2, 3]
const copiaArray = [...originalArray]

const array1 = [1, 2]
const array2 = [3, 4]
const combinado = [...array1, ...array2]
const combinado = array1.concat(array2)
```

También es útil con arrays de objetos.



Spread Operator / Rest Parameters

Rest Parameters ó parámetros rest, también representados por tres puntos consecutivos (...), permite a, por ejemplo, una función aceptar un número variable de argumentos y los recopila en un array.

Esto lo hace muy útil su aplicación en escenarios donde no sabemos cuántos argumentos se pasarán a una función, pero deseamos procesarlos de manera uniforme.



Spread Operator / Rest Parameters

Esto lo hace muy útil su aplicación en escenarios donde no sabemos cuántos argumentos se pasarán a una función, pero deseamos procesarlos de manera uniforme.

```
function sumar(...numeros) {  
    return numeros.reduce((total, numero) => total + numero, 0)  
}  
  
const resultado1 = sumar(1, 2, 3)           // resultado: 6  
const resultado2 = sumar(4, 5, 6, 7, 8)     // resultado: 30
```

¡Gracias!

educación IT