



## DWEC - Javascript Web Cliente.

# JavaScript 01 A – Sintaxis (I)

JavaScript 01 – Sintaxis .....	1
Variables .....	1
Use Strict .....	2
Otro ejemplo de ámbito de variables:.....	3
Variables locales y variables globales.....	4
Variables constantes (const) .....	5
Funciones.....	5
Parámetros.....	5
Funciones anónimas.....	7
Arrow functions (funciones flecha) .....	7
Estructuras y bucles.....	8
Estructura condicional: if .....	8
Estructura condicional: switch .....	9
Bucle while .....	9

## Variables

Javascript es un lenguaje débilmente tipado. Esto significa que no se indica de qué tipo es una variable al declararla, incluso puede cambiar su tipo a lo largo de la ejecución del programa. Ejemplo:

```
let miVariable;      // declaro miVariable y como no se asignó un valor valdrá undefined
miVariable='Hola';  // ahora su valor es 'Hola', por tanto contiene una cadena de texto
miVariable=34;       // pero ahora contiene un número
miVariable=[3, 45, 2]; // y ahora un array
miVariable=undefined; // para volver a valer el valor especial undefined
```

EJERCICIO: Ejecuta en la consola del navegador las instrucciones anteriores y comprueba el valor de **miVariable** tras cada instrucción (para ver el valor de una variable simplemente ponemos en la consola su nombre: `miVariable`

Ni siquiera estamos obligados a declarar una variable antes de usarla, aunque es recomendable para evitar errores que nos costará depurar.

Las variables se declaran con **let** (lo recomendado desde ES2015), aunque también pueden declararse con **var**.

La diferencia es que con **let** la variable sólo existe en el bloque en que se declara.

Mientras que con **var** el **ámbito (scope)** de la variable es global, se extiende. Es decir, existe en toda la función o ámbito en el que se declara:

```

if (edad > 18) {
    let textoLet = 'Eres mayor de edad';
    var textoVar = 'Eres mayor de edad';
} else {
    let textoLet = 'Eres menor de edad';
    var textoVar = 'Eres menor de edad';
}
console.log(textoLet); // mostrará undefined porque fuera del if no existe la variable
console.log(textoVar); // mostrará la cadena

```

Cualquier variable que no se declara dentro de una función (o si se usa sin declarar) es *global*. Debemos siempre intentar NO usar variables globales.

Además, como podemos comprobar con el siguiente ejemplo, las variables declaradas con var pueden duplicarse sin que se produzca error, y que a la larga puede acarrear muchos problemas.

```

// CREACIÓN DE VARIABLES CON LET

/* let declara una variable limitando su ámbito (scope) al bloque,
   declaración o expresión donde se está usando. */

// SINTAXIS: let nombreVariable [= valor];

var persona = "Santi";
var persona = "Profesor";
console.log (persona);

let persona2 = "Ana";
//let persona2 = "Jefa de Estudios"; //Esta instrucción devuelve un error
console.log (persona2);

```

Se recomienda que los nombres de las variables sigan la sintaxis *camelCase* (ej.: *miPrimeraVariable*).

Desde ES2015 también podemos declarar constantes con **const**. Se les debe dar un valor al declararlas y si intentamos modificarlo posteriormente se produce un error.

NOTA: en la página de [Babel](#) podemos teclear código en ES2015 y ver cómo quedaría una vez *transpilado* a ES5.

Cuando veamos “next generation JavaScript” se refiere a la próxima versión de Javascript pendiente de ser aceptada

## Use Strict

Para evitar problemas futuros, y que no se produzca algún error si no declaramos una variable, incluiremos al principio de nuestro código la instrucción “use estrict”, que nos obliga a declarar las variables.

```
'use strict';
```

Veamos un ejemplo:

```

// USE STRICT O MODO ESTRICTO

/* "use strict" es una línea que indica que el código debe ser usado "en modo estricto",
   es decir, no se pueden utilizar variables no declaradas.
   Fuera de una función tiene ámbito global; dentro de ella, local (el de la función).
*/

```

```
// SINTAXIS: "use strict";

// "use strict"; // si aplicamos este código da error al no declarar persona2

persona2 = "Santi";
let nacimiento;

function informar (){
    "use strict";
    let persona = "Santi";
    nacimiento = "1915";
    console.log(persona + " nacio en "+nacimiento);
}

informar();
```

Otro ejemplo de ámbito de variables:

**Caso 1º:** Usamos dos variables (a) definidas con let, cada una queda circunscrita al ámbito donde ha sido creada.

#### / ÁMBITO DE VARIABLES

```
/* El ámbito de una variable (scope) es la zona del programa en la que se define.
Javascript define dos ámbitos para variables: local y global.
Mediante var podemos definir como ámbito local el ámbito de una función.
Con let, por el contrario, podemos diferenciar también el ámbito de bloque. */
```

```
function verAmbito(){
    "use strict";
    let a = "Ámbito de función";
    if (true){
        let a = "Ámbito de bloque";
        console.log ("El ámbito de bloque a es: "+a);
    }
    console.log ("El ámbito de función a es: "+a);
}
verAmbito();
```

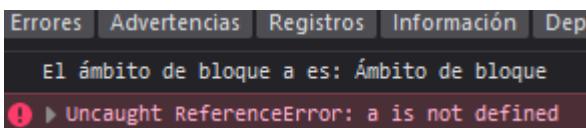
Errores Advertencias Registros Información Depura  
El ámbito de bloque a es: Ámbito de bloque  
El ámbito de función a es: Ámbito de función  
»

Si en el caso 1º cambiamos el **let** (dentro del if) por **var**, da error de variable ya declarada, puesto que ahora var extiende su scope a toda la función.

top ▾ | Filtrar  
✖ Uncaught SyntaxError: Identifier 'a' has already been declared

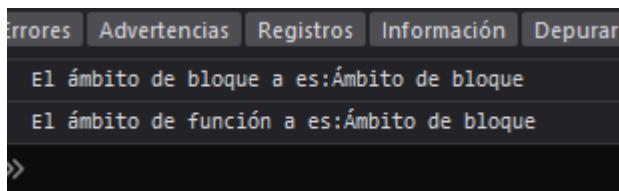
**Caso 2º:** Usamos una variable (a) definida con **let** dentro de un **if**, su ámbito queda reducido al bloque if, por tanto, fuera del bloque daría error.

```
function verAmbito(){
  "use strict";
  if (true){
    let a = "Ámbito de bloque";
    console.log ("El ámbito de bloque a es: "+a);
  }
  console.log ("El ámbito de función a es: "+a); // produce error
}
verAmbito();
```



**Caso 3º:** Usamos una variable (a) definida con **var** dentro de un **if**, su ámbito se extiende a toda la función.

```
function verAmbito(){
  "use strict";
  //let a = "Ámbito de función"; // SIMILAR AL CASO: var a = "Ámbito de función";
  if (true){
    var a = "Ámbito de bloque";
    console.log ("El ámbito de bloque a es:"+a);
  }
  console.log ("El ámbito de función a es:"+a);
}
verAmbito();
// console.log ("El ámbito de función a es: "+a); // ERROR, pues no llega el ámbito
```



Nota: si en el caso 3º cambiamos var por let (estamos igual que el caso)

**Conclusiones:** Se pueden trabajar con **var** y **let** a la vez, pero sería peligroso. En la medida de lo posible trabajaremos con **let**, e incluso con **const** cuando sepamos que esa variable nunca debe cambiar. Además, debemos pensar en que al usar **const** y **let** (su ámbito queda reducido a nuestros bloques) nunca interferirán con otras funciones o código de otro programador en nuestra aplicación.

A modo de repaso recordemos que en este lenguaje de programación no es obligado declarar las variables.

En el caso de ECMAScript 5 usamos la palabra reservada **var**.

En el caso de ECMAScript 6+ además de **var** también podemos usar **let** y **const**.

## Variables locales y variables globales

Una variable global se puede usar en cualquier lugar del script.

Una variable local sólo tiene validez dentro del ámbito de una función.

Las normas que aplican para definir el tipo de variable que estamos usando se resumen con 3 normas sencillas:

- Cualquier variable declarada o no declarada en la raíz de un script es siempre de ámbito global.
- Cualquier variable declarada dentro de una función es de ámbito local.
- Una variable NO declarada dentro de una función adquiere ámbito global.

En caso de duda, que es cuando la variable no ha sido declarada en un ámbito determinado, la variable adquiere comportamiento global.

### Casos de uso de las propiedades del ámbito de las variables

En javascript se pueden dar algunos casos curiosos:

- Dentro de una función podemos sobrescribir el valor de una variable global. Esto puede suceder por ejemplo por error si nos hemos olvidado de declarar una variable... y cargarnos un dato externo a la función sin darnos cuenta.
- Cuando declaramos una variable local con el mismo nombre que una variable global, temporalmente la variable local tiene prioridad sobre la global. Dentro de la función usaremos la variable local. Deberemos tratarlas como si temporalmente la variable global hubiera dejado de existir para nosotros.
- Hay que recordar que una variable declarada dentro de los paréntesis de declaración de la función se considera declarada de tipo local y que por cuestiones de sintaxis nunca va acompañada de la palabra reservada `let`.

### Variables constantes (`const`)

Las variables constantes en Javascript (`const`) tienen ámbito de bloque al igual que las variables definidas utilizando `let`. Es importante tener en cuenta que el valor de una constante no puede variar (reasignarse), por tanto, se asignan en el momento en que se declaran. Para diferenciarlo de las variables conviene utilizar **TODOMAYÚSCULAS**.

## Funciones

Se declaran con la palabra reservada **function** y se les pasan los parámetros entre paréntesis. La función puede devolver un valor usando **return** (si no tiene `return` es como si devolviera `undefined`).

Puede usarse una función antes de haberla declarado por el comportamiento de Javascript llamado *hoisting*: el navegador primero carga todas las funciones y mueve las declaraciones de las variables al principio y luego ejecuta el código.

**EJERCICIO mensaje:** Realiza una función que te pida que escribas algo y muestre un alert diciendo 'Has escrito...' y el valor introducido. Pruébala en la consola (pegas allí la función y luego la llamas desde la consola)

### Parámetros

Si se llama a una función con menos parámetros de los declarados, el valor de los parámetros no pasados será `undefined`:

```
function potencia(base, exponente) {
    console.log(base);           // muestra 4
    console.log(exponente);      // muestra undefined
    let valor=1;
```

```

    for (let i=1; i<=exponente; i++) {
        valor=valor*base;
    }
    return valor;
}

let resultado = potencia(4); // devolverá 1 ya que no se ejecuta el for
console.log(resultado);
//console.log(potencia(4)); // devolverá 1 ya que no se ejecuta el for

```

Podemos dar un **valor por defecto** a los parámetros por si no los pasan asignándoles el valor al definirlos:

```

function potencia(base, exponente=2) {
    console.log(base);           // muestra 4
    console.log(exponente);      // muestra 2 la primera vez y 5 la segunda
    let valor=1;
    for (let i=1; i<=exponente; i++) {
        valor=valor*base;
    }
    return valor;
}

console.log(potencia(4));       // mostrará 16 (4^2)
console.log(potencia(4,5));     // mostrará 1024 (4^5)

```

NOTA: En ES5 para dar un valor por defecto a una variable se hacía:

```

function potencia(base, exponente) {
    exponente = exponente || 2; // si exponente vale undefined se la asigna el valor 2
    ...
}

```

También es posible acceder a los parámetros desde el array **arguments[]** por si no sabemos cuántos parámetros recibiremos:

```

function sumar () {
    let result = 0;
    for (let i=0; i<arguments.length; i++)
        result += arguments[i];
    return result;
}

console.log(sumar(4, 2));           // mostrará 6
console.log(sumar(4, 2, 5, 3, 2, 1, 3)); // mostrará 20

```

En Javascript las funciones son un tipo de datos más, por lo que podemos hacer cosas como pasárlas como argumento o asignárlas a una variable:

```

const cuadrado=function(value){
    return value * value;
}
function aplicarFuncion(dato, funcion_a_aplicar){

```

```

        return funcion_a_aplicar(dato);
    }
let resultado = aplicarFuncion(3,cuadrado); // devolverá 9 (3^2) y lo guarda en resultado

```

A este tipo de funciones (en nuestro caso la función *cuadrado*) se llaman *funciones de primera clase* y son típicas de lenguajes funcionales.

Un lenguaje de programación se dice que tiene **Funciones de primera clase** cuando las funciones en ese lenguaje son tratadas como cualquier otra variable. Por ejemplo, en ese lenguaje, una función puede ser pasada como argumento a otras funciones, puede ser retornada por otra función y puede ser asignada a una variable.

## Funciones anónimas

Como acabamos de ver, podemos definir una función sin darle un nombre. Dicha función puede asignarse a una variable, autoejecutarse o asignarse a un manejador de eventos. Ejemplo:

```

let holaMundo = function() {
    alert('Hola mundo!');
}

holaMundo();           // se ejecuta la función

```

Como vemos, asignamos una función a una variable de forma que podamos “ejecutar” dicha variable.

## Arrow functions (funciones flecha)

ES2015 permite declarar una función anónima de forma más corta. Ejemplo sin *arrow function*:

```

let potencia = function(base, exponente) {
    let valor=1;
    for (let i=1; i<=exponente; i++) {
        valor=valor*base;
    }
    return valor;
}

```

Al escriirla con la sintaxis de una *arrow function* lo que hacemos es:

- Eliminamos la palabra *function*
- Si sólo tiene 1 parámetro podemos eliminar los paréntesis de los parámetros
- Ponemos el símbolo =>
- Si la función sólo tiene 1 línea podemos eliminar las {} y la palabra *return*

El ejemplo con *arrow function*:

```

let potencia = (base,exponente) => {
    let valor=1;
    for (let i=1; i<= exponente; i++){
        valor= valor * base
    }
    return valor
}

```

Otra forma de la función anterior con *arrow function*:

```
let resultado = (base, exponente) => base ** exponente;
```

Otro ejemplo, sin *arrow function*:

```
let cuadrado= function(base) {
    return base * base;
}
```

con *arrow function*:

```
let cuadrado = (base) => base * base; //puedes eliminar los paréntesis porque hay un único argumento
```

EJERCICIO: Haz una *arrow function* que devuelva el cubo del número pasado como parámetro y pruébala desde la consola. Escríbela primero en la forma habitual y luego la “traduces” a *arrow function*.

## Estructuras y bucles

### Estructura condicional: if

El **if** es como en la mayoría de los lenguajes. Puede tener asociado un **else** y pueden anidarse varios con **else if**.

```
if (condicion) {
    ...
} else if (condicion2) {
    ...
} else if (condicion3) {
    ...
} else {
    ...
}
```

Ejemplo:

```
if (edad < 18) {
    console.log('Es menor de edad');
} else if (edad > 65) {
    console.log('Está jubilado');
} else {
    console.log('Edad correcta');
}
```

### Operador ternario

Se puede usar el operador ternario **? :** que es como un *if* que devuelve un valor:

```
let edad=14;
let esMayorDeEdad = edad > 18 ? true : false;
console.log(esMayorDeEdad);
```

## Estructura condicional: switch

El **switch** también es como en la mayoría de los lenguajes. Hay que poner *break* al final de cada bloque para que no continúe evaluando:

```
switch(color) {
    case 'blanco':
    case 'amarillo': // Ambos colores entran aquí
        colorFondo='azul';
        break;
    case 'azul':
        colorFondo='amarillo';
        break;
    default:          // Para cualquier otro valor
        colorFondo='negro';
}
```

Javascript permite que el *switch* en vez de evaluar valores pueda evaluar expresiones. En este caso se pone como condición *true*:

```
switch(true) {
    case age < 18:
        console.log('Eres muy joven para entrar');
        break;
    case age < 65:
        console.log('Puedes entrar');
        break;
    default:
        console.log('Eres muy mayor para entrar');
}
```

## Bucle while

Podemos usar el bucle *while...do*

```
while (condicion) {
    // sentencias
}
```

que se ejecutará 0 o más veces. Ejemplo:

```
let nota = prompt("Introduce una nota (o cancela para finalizar)");
while (nota) {
    console.log("La nota introducida es: " + nota);
    nota = prompt("Introduce una nota (o cancela para finalizar");
}
```

Si queremos asignar más de una **condición** al bucle, usaremos los operadores lógicos. En el siguiente ejemplo, el bucle finaliza cuando se pulsa cancelar o se introduce el valor **cero**.

```
let nota = prompt("Introduce una nota (o cancela para finalizar)");
while (nota && nota!=0) {
```

```

    console.log("La nota introducida es: " + nota);
    nota = prompt("Introduce una nota (o cancela para finalizar)");
}
console.log("última nota: "+ nota);

```

O el bucle do...while:

```

do {
    // sentencias
} while (condicion)

```

que al menos se ejecutará 1 vez. Ejemplo:

```

let nota;
do {
    nota=prompt('Introduce una nota (o cancela para finalizar)');
    console.log('La nota introducida es: '+nota);
} while (nota)

```

**EJERCICIO adivina:** Haz un programa para que el usuario juegue a adivinar un número. Obtén un número al azar (busca por internet cómo se hace o simplemente guarda el número que quieras en una variable) y ve pidiendo al usuario que introduzca un número. Si es el que busca, le dices que lo ha encontrado y si no le mostrarás si el número que busca es mayor o menor que el introducido. El juego acaba cuando el usuario encuentra el número o cuando pulsa en ‘Cancelar’ (en ese caso le mostraremos un mensaje de que ha cancelado el juego).



## DWEC - Javascript Web Cliente.

# JavaScript 01 B – Sintaxis (II)

JavaScript 01 B – Sintaxis (II).....	1
Bucle: for.....	1
Bucle: for con contador.....	1
Bucle: for...in .....	1
Bucle: for...of .....	2

### Bucle: for

Tenemos muchos *for* que podemos usar.

#### *Bucle: for con contador*

Creamos una variable **contador** que controla las veces que se ejecuta el *for*:

```
let datos=[5, 23, 12, 85]
let sumaDatos=0;

for (let i=0; i<datos.length; i++) {
    sumaDatos += datos[i];
}
// El valor de sumaDatos será 125
```

EJERCICIO: El factorial de un número entero n es una operación matemática que consiste en multiplicar ese número por todos los enteros menores que él: **n x (n-1) x (n-2) x ... x 1**. Así, el factorial de 5 (se escribe 5!) vale **5! = 5 x 4 x 3 x 2 x 1 = 120**. Haz un script que calcule el factorial de un número entero positivo.

#### *Bucle: for...in*

El bucle se ejecuta una vez para cada elemento del array (o propiedad del objeto) y se crea una variable **índice** que toma como valores la posición del elemento en el array:

```
let datos=[5, 23, 12, 85]
let sumaDatos=0;

for (let indice in datos) {
    sumaDatos += datos[indice];      // Los valores que toma indice son 0, 1, 2, 3
}
// El valor de sumaDatos será 125
```

También sirve para recorrer las propiedades de un objeto:

```

let profe={
    nom:'Santi',
    ape1:'Blanco',
    ape2:'Arenal'
}

let nombreCompleto='';

for (let campo in profe) {
    nombreCompleto += profe[campo] + ' ';
}
console.log(nombreCompleto);
// El valor de nombreCompleto será 'Santiago Blanco Arenal'

```

### Bucle: for...of

Es similar al `for...in` pero la variable contador en vez de tomar como valor cada índice toma cada elemento. Es nuevo en ES2015:

```

let sumaDatos = 0;

for (let valor of datos) {
    sumaDatos += valor;      // los valores que toma valor son 5, 23, 12, 85
}
// El valor de sumaDatos será 125

```

También sirve para recorrer los caracteres de una cadena de texto:

```

let cadena = 'Hola';

for (let letra of cadena) {
    console.log(letra);      // los valores de letra son 'H', 'o', 'l', 'a'
}

```

EJERCICIO: Haz 3 funciones a las que se le pasa como parámetro un array de notas y devuelve la nota media. Cada una usará un `for` de una de las 3 formas vistas. Pruébalas en la consola.



## DWEC - Javascript Web Cliente.

# JavaScript 01 – Sintaxis (III)

JavaScript 01 – Sintaxis (III) .....	1
Tipos de datos básicos .....	1
Casting de variables .....	2
Number .....	2
Veamos algunos ejemplos de código y la salida en consola:.....	3
Otras funciones útiles son:.....	4
String.....	4
Template literals .....	5
Tratamiento de una cadena como un array .....	5
Boolean .....	6

## Tipos de datos básicos

Para saber de qué tipo es el valor de una variable tenemos el operador **typeof**. Ej.:

- `typeof 3` devuelve *number*
- `typeof 'Hola'` devuelve *string*

En Javascript hay 2 valores especiales:

- **undefined**: es lo que vale una variable a la que no se ha asignado ningún valor.
- **null**: es un tipo de valor especial que podemos asignar a una variable. Es como un objeto vacío (`typeof null` devuelve *object*)

También hay otros valores especiales relacionados con operaciones numéricas (o con números):

- **NaN (Not a Number)**: indica que el resultado de la operación no puede ser convertido a un número (ej. `'Hola'*2`, aunque `'2'*2` daría 4 ya que se convierte la cadena '2' al número 2)
- **Infinity y -Infinity**: indica que el resultado es demasiado grande o demasiado pequeño (ej. `1/0` o `-1/0`)

```
js > JS funciones.js > ...
You, hace 33 segundos | 2 authors (You and others)
1 'use strict'; Profesor, anteal
2 console.log("tareaxxx");
3 console.log(typeof(3));
4 console.log(typeof("pepe"));
5 let a;
6 console.log(typeof(a));
7 console.log(typeof(null));
8 console.log("hola"*2);
9 console.log("3"*2);
10 console.log(-2/0);
11 console.log(typeof(-2/0));
12
```

Group similar messages in console    Show CORS errors in console    Treat code evaluation as user action

tareaxxx	funciones.js:2
number	funciones.js:3
string	funciones.js:4
undefined	funciones.js:6
object	funciones.js:7
NaN	funciones.js:8
6	funciones.js:9
-Infinity	funciones.js:10
number	funciones.js:11

## Casting de variables

Como hemos dicho, las variables pueden contener cualquier tipo de valor.

En las operaciones, Javascript realiza **automáticamente** las conversiones necesarias para, si es posible, realizar la operación.

Ejemplos:

- '4' / 2 → devuelve 2 (convierte '4' en 4 y realiza la operación)
- '23' - null → devuelve 23 (hace 23 - 0)
- '23' - undefined → devuelve NaN (no puede convertir undefined a nada, así que no puede hacer la operación)
- '23' \* true → devuelve 23 (23 \* 1)
- '23' \* 'Hello' → devuelve NaN (no puede convertir 'Hello')
- 23 + 'Hello' → devuelve '23Hello' (+ es el operador de concatenación, así que convierte 23 a '23' y los concatena)
- 23 + '23' → devuelve 2323 (OJO, convierte 23 a '23', no al revés)

Ten en cuenta que en Javascript todo son objetos, por lo que todo tiene métodos y propiedades. Veamos brevemente los tipos de datos básicos.

**EJERCICIO:** Prueba en la consola las operaciones anteriores y alguna más con la que tengas dudas de qué devolverá.

## Number

Sólo hay 1 tipo de números, no existen enteros y decimales. El tipo de dato para cualquier número es **number**. El carácter para la coma decimal es el . (como en inglés, así que 23,12 debemos escribirlo como 23.12).

Tenemos los operadores aritméticos +, -, \*, /, \*\* y % y los unarios ++ y --

Existen los valores especiales **Infinity** y **-Infinity** ( 23/0 no produce un error sino que devuelve *Infinity*).

Podemos usar los operadores aritméticos junto al operador de asignación = (=, -=, \*=, /= y %=).

Cuidado!! Observa la diferencia en los ejemplos de abajo.

The screenshot shows two examples of arithmetic assignment operators in a browser's developer tools console. In the first example, line 14 contains a syntax error (x+=y; instead of x=y+), resulting in 'You, hace 3 segundos'. The output 'NaN' is highlighted with a red box. In the second example, line 13 has 'let x=0;' highlighted with a red box, and the output '5' is also highlighted with a red box.

```

12 let y=5;
13 let x;
14 x+=y;      You, hace 3 segundos
15 console.log(x);          NaN
                                         funciones.js:15

12 let y=5;
13 let x=0;           Group similar messages in console
14 x+=y;             Show CORS errors in console
15 console.log(x);    Treat code evaluation as user action
                                         >                                funciones.js:15
                                         5
                                         
```

Algunos métodos útiles de los números son:

- **.toFixed(num)**: redondea el número a los decimales indicados. Ej. 23.2376.toFixed(2) devuelve 23.24
- **.toLocaleString()**: devuelve el número convertido al formato local. Ej. 23.76.toLocaleString() devuelve '23,76' (Los navegadores en español convierten el punto decimal en coma y ponen el punto separador de miles)

16	<code>let x=12536.231265;</code>	12536.23	<a href="#">funciones.js:17</a>
17	<code>console.log (x.toFixed(2));</code>		
18	<code>console.log (x.toLocaleString());</code>	12.536,231	<a href="#">funciones.js:18</a>

Podemos forzar la conversión a número con la función **Number(valor)**. Ejemplo `Number('23.12')` devuelve 23.12

Veamos algunos ejemplos de código y la salida en consola:

```
let num=23.33333333333;
console.log(typeof(num));
console.log(num);
```

```
num= 45212.44444
console.log(typeof(num));
console.log(num.toLocaleString());
```

```
let num2 = 36.2222;
console.log(typeof(num2));
num2=num2.toLocaleString();
console.log(typeof(num2));
console.log(num2);
```

```
num2=888; //ojo
console.log(num2);
num2=num2.toLocaleString();
console.log(num2);
console.log(typeof(num2));
num2 = Number(num2);
console.log(typeof(num2));
console.log(num2);
```

```
num2=888.8888; //ojo
console.log(num2);
num2=num2.toLocaleString();
console.log(num2);
console.log(typeof(num2));
num2 = Number(num2);
console.log(typeof(num2));
console.log(num2);
```

En el caso anterior, no es numérico porque después de convertir `toLocaleString()`, no puede volver a numérico.

Si se convierte con el método `toString()`, sí que puede retornar a numérico, como vemos en el siguiente ejemplo:

The screenshot shows a browser's developer tools console. On the left, there is a block of JavaScript code:

```

num2=888.8888; //ojo
console.log(num2);
num2=num2.toString();
console.log(num2);
console.log(typeof(num2));
num2 = Number(num2);
console.log(typeof(num2));
console.log(num2);

```

Red arrows point from the highlighted line (`num2=num2.toString();`) to the output lines. The first two outputs are "888.8888". The third output is "string". The fourth output is "number", which is also highlighted with a red box. The fifth output is "888.8888". The final output is ">".

Otras funciones útiles son:

- isNaN(valor)**: nos dice si el valor pasado es un número (false), o no (true)
- isFinite(valor)**: devuelve *true* si el valor es finito (no es *Infinity* ni *-Infinity*).
- parseInt(valor)**: convierte el valor pasado a un número entero. Siempre que comience por un número la conversión se podrá hacer. Ej.:

```

parseInt(3.65)      // Devuelve 3
parseInt('3.65')    // Devuelve 3
parseInt('3 manzanas') // Devuelve 3, Number devolvería NaN

```

- parseFloat(valor)**: como la anterior, pero conserva los decimales

**OJO**: al sumar números decimales (*floats*) podemos tener problemas:

```
console.log(0.1 + 0.2) // imprime 0.30000000000000004
```

Para evitarlo:

- Redondead los resultados.
- También se puede hacer una operación similar a  $(0.1*10 + 0.2*10) / 10$ .

console.log((0.1 + 0.2));	0.30000000000000004
console.log((0.1 + 0.2).toFixed(1));	0.3
console.log( (0.1*10 + 0.2*10) / 10);	0.3

EJERCICIO: Modifica la función de calcular la nota media para que devuelva la media con 1 decimal

EJERCICIO: Modifica la función que devuelve el cubo de un número para que compruebe si el parámetro pasado es un número entero. Si no es un entero o no es un número mostrará un alert indicando cuál es el problema y devolverá false.

## String

Las cadenas de texto van entre comillas simples o dobles, es indiferente. Podemos escapar un carácter con \ para poder usarlo dentro de la cadena. (Ejemplo: 'Hola \'Mundo\' ' devuelve Hola 'Mundo').

console.log("hola \"majo\"");	hola "majo"
console.log('Hola "majo"');	Hola "majo"

Para forzar la conversión a cadena se usa la función **String(valor)** (ej. `String(23)` devuelve '23')

El operador de concatenación de cadenas es `+`. Ojo porque si pedimos un dato con `prompt` siempre devuelve una cadena así que si le pedimos la edad al usuario (por ejemplo 20) y se sumamos 10 tendremos 2010 ('20'+10).

Algunos métodos y propiedades de las cadenas son:

- **`.length`**: devuelve la longitud de una cadena. Ej.: 'Hola mundo'.length devuelve 10
- **`.charAt(posición)`**: 'Hola mundo'.charAt(0) devuelve 'H'
- **`.indexOf(carácter)`**: 'Hola mundo'.indexOf('o') devuelve 1. Si no se encuentra devuelve -1
- **`.lastIndexOf(carácter)`**: 'Hola mundo'.lastIndexOf('o') devuelve 9
- **`.substring(desde, hasta)`**: 'Hola mundo'.substring(2,4) devuelve 'la'
- **`.substr(desde, num caracteres)`**: 'Hola mundo'.substr(2,4) devuelve 'la m'
- **`.replace(busco, reemplaza)`**: 'Hola mundo'.replace('Hola', 'Adiós') devuelve 'Adiós mundo'
- **`.toLocaleLowerCase()`**: 'Hola mundo'.toLocaleLowerCase() devuelve 'hola mundo'
- **`.toLocaleUpperCase()`**: 'Hola mundo'.toLocaleUpperCase() devuelve 'HOLA MUNDO'
- **`.localeCompare(cadena)`**: devuelve -1 si la cadena a que se aplica el método es anterior alfabéticamente a 'cadena', 1 si es posterior y 0 si ambas son iguales. Tiene en cuenta caracteres locales como acentos ñ, ç, etc
- **`.trim(cadena)`**: ' Hola mundo '.trim() devuelve 'Hola mundo'
- **`.startsWith(cadena)`**: 'Hola mundo'.startsWith('Hol') devuelve true
- **`.endsWith(cadena)`**: 'Hola mundo'.endsWith('Hol') devuelve false
- **`.includes(cadena)`**: 'Hola mundo'.includes('mun') devuelve true
- **`.repeat(veces)`**: 'Hola mundo'.repeat(3) devuelve 'Hola mundoHola mundoHola mundo'
- **`.split(separador)`**: 'Hola mundo'.split(' ') devuelve el array ['Hola', 'mundo']. 'Hola mundo'.split('') devuelve el array ['H', 'o', 'l', 'a', '', 'm', 'u', 'n', 'd', 'o']

Podemos probar los diferentes métodos en la página de [w3schools](#).

### Template literals

Desde ES2015 también podemos poner una cadena entre ` (acento grave) y en ese caso podemos poner dentro variables y expresiones que serán evaluadas al ponerlas dentro de  `${}`. También se respetan los saltos de línea, tabuladores, etc que haya dentro. Ejemplo:

```
let edad=25;
console.log(`El usuario tiene:
${edad} años`);
```

```
El usuario tiene:
25 años
```

### Tratamiento de una cadena como un array

The screenshot shows a browser's developer tools with the 'Console' tab selected. On the left, a code block is shown:

```
let cadena = "abcdefg";
console.log(`Cadena es el array: ${cadena}`);
console.log(`Valor en la posición 0: ${cadena[0]}`);
console.log(`Valor en la posición 1: ${cadena[1]}`);
console.log(`Valor en la posición última: ${cadena[cadena.length - 1]}`);
for (const letra of cadena) {
    console.log(letra);
}
```

Red arrows point from the console.log statements to their corresponding outputs in the console. A red box highlights the entire code block. Another red box highlights the output array [a, b, c, d, e, f, g].

The console output is:

- Cadena es el array: abcdefg
- Valor en la posición 0: a
- Valor en la posición 1: b
- Valor en la posición última: g
- a
- b
- c
- d
- e
- f
- g

**EJERCICIO:** Haz una función a la que se le pasa un DNI (ej. 12345678w o 87654321T) y devolverá si es correcto o no. La letra que debe corresponder a un DNI correcto se obtiene dividiendo la parte numérica entre 23 y cogiendo de la cadena 'TRWAGMYFPDXBNJZSQVHLCKE' la letra correspondiente al resto de la división. Por ejemplo, si el resto es 0 la letra será la T y si es 4 será la G. Prueba la función en la consola con tu DNI

## Boolean

Los valores booleanos son **true** y **false**. Para convertir algo a booleano se usar **Boolean(valor)** aunque también puede hacerse con la doble negación (**!!**). Cualquier valor se evaluará a *true* excepto 0, NaN, null, undefined o una cadena vacía ("") que se evaluarán a *false*.

The screenshot shows a browser's developer tools console with several Boolean evaluations:

```
console.log(Boolean("pepe"));
console.log{!!("pepe")};
console.log(!("pepe"));
console.log(!null);
console.log (!!null);
console.log (!!" ");
console.log (!!(""));
console.log (!!undefined);
```

Red arrows point from each expression to its result. A red box highlights the first two rows.

The results are:

Expresión	Resultado
Boolean("pepe")	true
{!!("pepe")}	true
!("pepe")	false
!null	true
(!!null)	false
(!!" ")	true
(!!(""))	false
(!!undefined)	false

Los operadores lógicos son ! (negación), && (and), || (or).

Para comparar valores tenemos == y ===. La triple igualdad devuelve *true* si son de igual valor y del mismo tipo. Como Javascript hace conversiones de tipos automáticas conviene usar la === para evitar cosas como:

- '3' == 3 true
- 3 == 3.0 true
- 0 == false true
- '' == false true
- '' == false true
- [] == false true
- null == false false
- undefined == false false
- undefined == null true

También tenemos 2 operadores lógicos para *diferente*: != y !== que se comportan como hemos dicho antes.

Los operadores relacionales son >, >=, <, <=. Cuando se compara un número y una cadena, ésta se convierte a número y no al revés (23 > '5' devuelve *true*, aunque '23' > '5' devuelve *false*)

console.log( 23 > 5 );	true
console.log( '23' > '5' );	false
console.log( '23' > 5 );	true
console.log( 23 > '5' );	true
.	.



## DWEC – Javascript Web Cliente.

# JavaScript 01 – Sintaxis (IV)

JavaScript 01 – Sintaxis (IV).....	1
Manejo de errores .....	1
Buenas prácticas .....	4
'use strict' .....	4
Variables .....	4
Otras .....	4
Clean Code.....	5

## Manejo de errores

Si sucede un error en nuestro código el programa dejará de ejecutarse, por lo que el usuario tendrá la sensación de que no hace nada (el error sólo se muestra en la consola y el usuario no suele abrirla nunca). Para evitarlo debemos intentar capturar los posibles errores de nuestro código antes de que se produzcan.

Pero hay una construcción sintáctica **try...catch** que nos permite “atrapar” errores para que el script pueda, en lugar de morir, hacer algo más razonable.

```
try {
    // código...
} catch (err) {
    // manipulación de error
}
```

Dentro del bloque *try* ponemos el código que queremos proteger y cualquier error producido en él será pasado al bloque *catch* donde es tratado. Opcionalmente podemos tener al final un bloque *finally* que se ejecuta tanto si se produce un error como si no.

Funciona así:

- Primero, se ejecuta el código en *try { ... }*.
- Si no hubo errores, se ignora *catch (err)*: la ejecución llega al final de *try* y continúa, omitiendo *catch*.
- Si se produce un error, la ejecución de *try* se detiene y el control fluye al comienzo de *catch (err)*.
- La variable *err* (podemos usar cualquier nombre para ella) contendrá un objeto de error con detalles sobre lo que sucedió.

El parámetro que recibe *catch* es un objeto con las propiedades *name*, que indica el tipo de error (*SyntaxError*, *RangeError*, ... o el genérico *Error*), y *message*, que indica el texto del error producido.

```
try {
  jaja
} catch (err) {
  console.log(err);
  console.log(err.name);
  console.log(err.message);
}
```

ReferenceError: jaja is not defined  
at funciones.js:10:3  
ReferenceError  
jaja is not defined

Para que `try..catch` funcione, el código debe ser ejecutable. En otras palabras, debería ser JavaScript válido.

No funcionará si el código es sintácticamente incorrecto, por ejemplo, si hay llaves sin cerrar.

El motor de JavaScript primero lee el código y luego lo ejecuta. Los errores que ocurren en la fase de lectura se denominan errores de “tiempo de análisis” y son irrecuperables (desde dentro de ese código). Eso es porque el motor no puede entender el código.

Entonces, `try...catch` solo puede manejar errores que ocurren en un código válido. Dichos errores se denominan “errores de tiempo de ejecución” o, a veces, “excepciones”.

En ocasiones podemos querer que nuestro código genere un error. Esto evita que tengamos que comprobar si el valor devuelto por una función es el adecuado o es un código de error.

### Sin usar `try..catch`

```
1 jaja;
2 console.log("no sigue");
3
```

Uncaught ReferenceError: jaja is not defined  
at funciones.js:1:1

En este caso no seguiría la ejecución (se para la aplicación) y la línea 2 no se ejecuta.

### Usando `try..catch`

```
1 try {
  2   jaja;
3 } catch (error) {
4   console.error(error);
5 }
6 console.log("Ahora sí continúa la ejecución");
7
```

ReferenceError: jaja is not defined  
at funciones.js:2:5  
Ahora sí continúa la ejecución

### Ejemplo de excepción definida por el programador:

Tenemos una función para retirar dinero de una cuenta que recibe:

- el saldo de la cuenta
- y la cantidad de dinero a retirar

La función devuelve:

- el nuevo saldo, pero si no hay suficiente saldo no debería restar nada sino mostrar un mensaje al usuario.

Si no gestionamos los errores haríamos:

```
function retirar(saldo, cantidad) {
  if (saldo < cantidad) {
    return false;
  }
  return saldo - cantidad;
```

```

}

let saldo = 30;
cantidad = 200;
let resultado = retirar(saldo, cantidad);

if (resultado === false) {
  console.log("Saldo insuficiente");
} else {
  saldo = resultado;
}

```

Se trata de un código poco claro que podemos mejorar lanzando un error en la función. Para ello se utiliza la instrucción `throw` que tiene la sintaxis: `throw expresión;`

```

if (saldo < cantidad) {
  throw 'Saldo insuficiente'
}

```

Por defecto al lanzar un error, este será de clase `Error` pero (el código anterior es equivalente a `throw new Error('Valor no válido')`) aunque podemos lanzarlo de cualquier otra clase (`throw new RangeError('Saldo insuficiente')`) o personalizarlo.

Siempre que vayamos a ejecutar código que pueda generar un error debemos ponerlo dentro de un bloque `try`.

Por lo que la llamada a la función que contiene el código anterior debería estar dentro de un `try`. El código del ejemplo anterior quedaría:

```

function retirar(saldo, cantidad) {
  if (saldo < cantidad) {
    // console.log("No hay saldo"); // esta línea no evita resultado erróneo
    throw "Saldo insuficiente";
  }
  return saldo - cantidad;
}

// Siempre debemos llamar a esa función desde un bloque _try_

let saldo = 30;
let importe = 200;

try {
  saldo = console.log(`Nuevo saldo: ${retirar(saldo, importe)}`);
} catch (err) {
  console.log(err); // muestra "Saldo Insuficiente"
}

try {
  saldo = console.log(`Nuevo saldo: ${retirar(200, 5)}`); //Muestra "Nuevo Saldo: 195"
} catch (err) {
  console.log(err);
}

```

Podemos ver en detalle cómo funcionan en la página de [MDN web docs](#) y <https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Statements/throw> de Mozilla.

### Try...catch trabaja sincrónicamente

Si ocurre una excepción en el código “programado”, como en `setTimeout`, entonces `try..catch` no lo detectará.

Más información en: <https://es.javascript.info/try-catch>

EJERCICIO: Escribe una función que devuelva la nota media de un array de notas. Si el array está vacío debe avisar del error. Prueba la función con `try catch`:

## Buenas prácticas

Javascript nos permite hacer muchas cosas que otros lenguajes no nos dejan, por lo que debemos ser cuidadosos para no cometer errores de los que **no se nos va a avisar**.

### ‘use strict’

Si ponemos siempre esta sentencia al principio de nuestro código el intérprete nos avisará si usamos una variable sin declarar (muchas veces por equivocarnos al escribir su nombre). En concreto fuerza al navegador a no permitir:

- Usar una variable sin declarar
- Definir más de 1 vez una propiedad de un objeto
- Duplicar un parámetro en una función
- Usar números en octal (comienzan por 0)
- Modificar una propiedad de sólo lectura

## Variables

Algunas de las prácticas que deberíamos seguir respecto a las variables son:

- Elegir un buen nombre es fundamental. Evitar abreviaturas o nombres sin significado (a, b, c, ...)
- Evitar en lo posible variables globales
- Usar `let` para declararlas
- Usar `const` siempre que una variable no deba cambiar su valor
- Declarar todas las variables al principio
- Inicializar las variables al declararlas
- Evitar conversiones de tipo automáticas
- Usar, para nombrarlas, la notación `camelCase`

También es conveniente, por motivos de eficiencia no usar objetos `Number`, `String` o `Boolean`, si no los tipos primitivos: (no usar `let numero = new Number(5)`, sino `let numero = 5`) y lo mismo al crear arrays, objetos o expresiones regulares (no usar `let miArray = new Array()`, sino `let miArray = []`).

## Otras

Algunas reglas más que deberíamos seguir son:

- Debemos ser coherentes a la hora de escribir código: por ejemplo, podemos poner (recomendado) o no espacios antes y después del `=` en una asignación, pero debemos hacerlo siempre igual. Existen muchas

guías de estilo y muy buenas: [Airbnb](#), [Google](#), [Idiomatic](#), etc. Para obligarnos a seguir las reglas podemos usar alguna herramienta [linter](#).

- También es conveniente para mejorar la legibilidad de nuestro código separar las líneas de más de 80 caracteres. Para saltar de línea en una cadena usamos el símbolo \

```
let cadena = "Esta es una cadena muy larga \
que me gustaría dividir en dos";
console.log (cadena);
//El resultado sería "Esta es una cadena muy larga que me gustaría dividir en dos";
```

- Usar === en las comparaciones
- Si un parámetro puede faltar al llamar a una función, darle un valor por defecto
- Y para acabar: **comentar el código** cuando sea necesario, pero mejor que el código sea lo suficientemente claro como para no necesitar comentarios

## Clean Code

Estas y otras muchas recomendaciones se recogen en el libro [Clean Code](#) de Robert C. Martin y en muchos otros libros y artículos. Aquí se puede ver un pequeño resumen traducido al castellano:

<https://github.com/devictoribero/clean-code-javascript>



## DWEC – Javascript Web Cliente.

JavaScript 02 – Arrays (I).....	2
Introducción.....	2
Objetos JavaScript.....	2
Propiedades de un objeto.....	2
Métodos de un objeto .....	3
Los arrays en JavaScript .....	4
Arrays de objetos .....	5
Operaciones con Arrays .....	5
length .....	5
Añadir elementos.....	5
Eliminar elementos .....	5
splice .....	6
slice .....	6
Arrays y Strings .....	6
sort.....	7
Otros métodos comunes.....	8
Functional Programming.....	9
filter.....	9
find .....	10
findIndex .....	11
every / some .....	11
map .....	11
reduce .....	11
forEach .....	14
includes .....	14
Array.from.....	14
Referencia vs Copia.....	15
Rest y Spread .....	16
Desestructuración de arrays .....	17
Los objetos Map y Set .....	18
Map.....	18
Set .....	19

# JavaScript 02 – Arrays (I)

## Introducción

### Objetos JavaScript

Los objetos son uno de los tipos de datos de JavaScript.

Los objetos se utilizan para almacenar colecciones de clave/valor (nombre/valor).

Se puede decir que un objeto JavaScript es una colección de valores con nombre.

En Javascript podemos definir cualquier variable como un objeto declarándola con **new** (NO se recomienda) o creando un *literal object* (usando notación **JSON**).

El siguiente ejemplo crea un objeto JavaScript con tres propiedades clave/valor.

Creando con *new* (no recomendado):

```
let alumno = new Object()
alumno.nombre = 'Carlos'      // se crea la propiedad 'nombre' y se le asigna un valor
alumno['apellidos'] = 'Pérez Ortiz' // se crea la propiedad 'apellidos'
alumno.edad = 19
```

Creando un *literal object* (es la forma **recomendada**). El ejemplo anterior quedaría:

```
let alumno = {
  nombre: 'Carlos',
  apellidos: 'Pérez Ortiz',
  edad: 19,
};
```

### Propiedades de un objeto

Podemos acceder a las propiedades con . (punto) o [ ]:

```
console.log(alumno.nombre)      // imprime 'Carlos'
console.log(alumno['nombre'])    // imprime 'Carlos'
let prop = 'nombre'
console.log(alumno[prop])       // imprime 'Carlos'
```

Si intentamos acceder a propiedades que no existen, no se produce un error: se devuelve *undefined*:

```
console.log(alumno.ciclo)        // muestra undefined
```

Sin embargo, se genera un error si intentamos acceder a propiedades de algo que no es un objeto:

```
console.log(alumno.ciclo)        // muestra undefined
console.log(alumno.ciclo.descrip) // se genera un ERROR
```

Para evitar ese error antes había que comprobar que existían las propiedades previas:

```
console.log(alumno.ciclo && alumno.ciclo.descrip)
```

```
// si alumno.ciclo es un objeto muestra el valor de
// alumno.ciclo.descrip. Si no muestra undefined
```

Con ES2020 (ES11) se ha incluido el operador `?.`  para evitar tener que comprobar esto nosotros:

```
console.log(alumno.ciclo?.descrip)
// si alumno.ciclo es un objeto muestra el valor de
// alumno.ciclo.descrip. Si no muestra undefined
```

Podremos recorrer las propiedades de un objeto con `for..in`:

```
for (let prop in alumno) {
    console.log(prop + ': ' + alumno[prop])
}
```

Nota: En este caso no se puede aplicar el bloque de código `for..of` porque un objeto de este tipo no es iterable, sí son iterables los arrays y strings

Si el valor de una propiedad es el valor de una variable que se llama como la propiedad, no es necesario ponerlo:

```
let nombre = 'Carlos'

let alumno = {
    nombre,           // es equivalente a nombre: nombre
    apellidos: 'Pérez Ortiz',
    ...
}
```

## Métodos de un objeto

Llamamos **método de un objeto** cuando una **propiedad** de ese objeto es una **función**:

```
alumno.getInfo = function() {
    return 'El alumno ' + this.nombre + ' ' + this.apellidos + ' tiene ' + this.edad + ' años'
}
```

Nota: La palabra clave `this` dentro de un método hace referencia al objeto en cuestión al que se aplica dicho método.

Y para llamarlo se hace como con cualquier otra propiedad:

```
console.log(alumno.getInfo()) // imprime 'El alumno Carlos Pérez Ortiz tiene 19 años'
```

Ojo!! No se puede utilizar `this` con sintaxis `arrow function`. Está explicado en el documento **Anexo – Uso de this en contexto**

## Algunos métodos y propiedades genéricas de objetos de JavaScript

- `constructor` Returns the function that created an object's prototype
- `keys()` Returns an Array Iterator object with the keys of an object
- `prototype` Let you to add properties and methods to JavaScript objects
- `toString()` Converts an object to a string and returns the result. Este método se usa sobreescrito.
- `valueOf()` Returns the primitive value of an object

Prueba las siguientes sentencias:

```
console.log(alumno.constructor);
console.log(Object.keys(alumno));
console.log(Object.values(alumno));
console.log(Object.entries(alumno));
console.log(alumno.toString());
alumno.edasd=33;
console.log(alumno.valueOf());
console.log(Object.prototype);
console.log(Object.constructor);
```

EJERCICIO: Crea un objeto llamado **tvSamsung** con las propiedades **nombre** ("TV Samsung 42"), **categoría** ("Televisores"), **unidades** (4), **precio** (345.95) y con un método llamado **importe** que devuelve el valor total de las unidades (nº de unidades \* precio).

## Los arrays en JavaScript

Los arrays, también llamados arreglos, vectores, matrices, listas..., son un tipo de objeto que no tiene tamaño fijo, podemos añadirle elementos en cualquier momento.

Para hacer referencia a (referenciar) los elementos, se hace con un índice numérico. A diferencia de los objetos que se referencian con un nombre.

Podemos crearlos como instancias del objeto Array (No se recomienda):

```
let a = new Array()          // a = []
let b = new Array(2,4,6)    // b = [2, 4, 6]
```

Lo recomendable es crearlos usando notación JSON:

```
let a = []
let b = [2,4,6]
```

Sus elementos pueden ser de cualquier tipo, incluso podemos tener elementos de tipos distintos en un mismo array.

Si no está definido un elemento, su valor será *undefined*. Ej.:

```
let a = ['Lunes', 'Martes', 2, 4, 6]
console.log(a[0]) // imprime 'Lunes'
console.log(a[4]) // imprime 6
a[7] = 'Juan'     // ahora a = ['Lunes', 'Martes', 2, 4, 6, , , 'Juan']
console.log(a[7]) // imprime 'Juan'
console.log(a[6]) // imprime undefined
```

Acceder a un elemento de un array que no existe no provoca un error (devuelve *undefined*), pero sí lo provoca acceder a un elemento de algo que no es un array. Con ES2020 (ES11) se ha incluido el operador **?** para evitar tener que comprobar nosotros que sea un array:

```
console.log(alumnos?[0])
// si alumnos es un array muestra el valor de su primer
// elemento y si no muestra undefined pero no lanza un error
```

## Arrays de objetos

Es habitual almacenar datos en arrays en forma de objetos, por ejemplo:

```
let alumnos = [
  {
    id: 1,
    name: 'Carlos Pérez',
    course: '2DAW',
    age: 21
  },
  {
    id: 2,
    name: 'Ana García',
    course: '2DAW',
    age: 23
  },
];
```

## Operaciones con Arrays

Los arrays tienen las mismas propiedades y métodos que los objetos, y muchos más que son propios de los arrays.

Vamos a ver los principales métodos y propiedades de los arrays.

### length

Esta propiedad devuelve la longitud de un array:

```
let a = ['Lunes', 'Martes', 2, 4, 6]
console.log(a.length) // imprime 5
```

Podemos reducir el tamaño de un array cambiando esta propiedad:

```
a.length = 3 // ahora a = ['Lunes', 'Martes', 2]
```

### Añadir elementos

Podemos añadir elementos al final de un array con el método `push`, o al principio con `unshift`:

```
let a = ['Lunes', 'Martes', 2, 4, 6]
a.push('Juan') // ahora a = ['Lunes', 'Martes', 2, 4, 6, 'Juan']
a.unshift(7) // ahora a = [7, 'Lunes', 'Martes', 2, 4, 6, 'Juan']
```

### Eliminar elementos

Podemos borrar el elemento del final de un array con `pop`, o del principio con `shift`. Ambos métodos devuelven el elemento que hemos borrado:

```
let a = ['Lunes', 'Martes', 2, 4, 6]
let ultimo = a.pop() // ahora a = ['Lunes', 'Martes', 2, 4] y ultimo = 6
let primero = a.shift() // ahora a = ['Martes', 2, 4] y primero = 'Lunes'
```

## splice

Permite eliminar elementos de cualquier posición del array y/o insertar otros en su lugar. Devuelve un array con los elementos eliminados. Sintaxis:

```
Array.splice(posicion, num. de elementos a eliminar, 1º elemento a insertar, 2º elemento a insertar, 3º...)
```

*Ejemplo:*

```
let a = ['Lunes', 'Martes', 2, 4, 6]
let borrado = a.splice(1, 3)           // ahora a = ['Lunes', 6] y borrado = ['Martes', 2, 4]
a = ['Lunes', 'Martes', 2, 4, 6]
borrado = a.splice(1, 0, 45, 56)      // ahora a = ['Lunes', 45, 56, 'Martes', 2, 4, 6] y borrado = []
a = ['Lunes', 'Martes', 2, 4, 6]
borrado = a.splice(1, 3, 45, 56)      // ahora a = ['Lunes', 45, 56, 6] y borrado = ['Martes', 2, 4]
```

EJERCICIO: Guarda en un array la lista de la compra con Peras, Manzanas, Kiwis, Plátanos y Mandarinas. Haz lo siguiente con splice:

- Elimina las manzanas (debe quedar Peras, Kiwis, Plátanos y Mandarinas)
- Añade detrás de los Plátanos, Naranjas y Sandía. (Debe quedar: Peras, Kiwis, Plátanos, Naranjas, Sandía y Mandarinas)
- Quita los Kiwis y pon en su lugar Cerezas y Nísperos. (Debe quedar: Peras, Cerezas, Nísperos, Plátanos, Naranjas, Sandía y Mandarinas)

## slice

Devuelve un subarray con los elementos indicados, pero sin modificar el array original. Sería como hacer un substr pero de un array en vez de una cadena.

Sintaxis: `Array.slice(posicion, num. de elementos a devolver)`

*Ejemplo:*

```
let a = ['Lunes', 'Martes', 2, 4, 6]
let subArray = a.slice(1, 3) // ahora a=[ 'Lunes', 'Martes', 2, 4, 6] y subArray=['Martes', 2, 4]
```

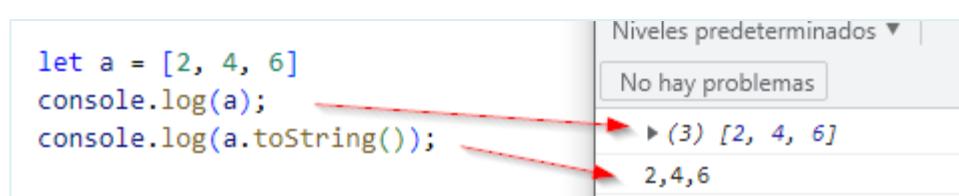
Es muy útil para hacer una copia de un array:

```
let a = [2, 4, 6]
let copiaDeA = a.slice() // ahora ambos arrays contienen lo mismo pero son diferentes arrays
```

## Arrays y Strings

Cada objeto, y los arrays son un tipo de objeto, tienen definido el método `.toString()` que lo convierte en una cadena. Este método es llamado automáticamente cuando, por ejemplo, queremos mostrar un array por la consola.

En el caso de los arrays, esta función devuelve una cadena con los elementos del array separados por coma.



Además, podemos convertir los elementos de un array a una cadena con `.join()` especificando el carácter separador de los elementos. Ej.:

```
let a = ['Lunes', 'Martes', 2, 4, 6]
let cadena = a.join('-') // cadena = 'Lunes-Martes-2-4-6'
```

El método `.join()` es el contrario del `.split()` que convierte una cadena en un array. Ej.:

```
let notas = '5-3.9-6-9.75-7.5-3'
let arrayNotas = notas.split('-') // arrayNotas = ['5', '3.9', '6', '9.75', '7.5', '3']
let cadena = 'Que tal estás'
let arrayPalabras = cadena.split(' ') // arrayPalabras = ['Que', 'tal', 'estás']
let arrayLetras = cadena.split('') // arrayLetras = ['Q','u','e`,' ','t',a',l',' ','e',s',t',á',s']
```

## sort

Ordena **alfabéticamente** los elementos del array.

OJO!! El array original queda modificado con el nuevo orden.

```
let a = ['hola', 'adios', 'Bien', 'Mal', 2, 5, 13, 45]
let b = a.sort() // b = [13, 2, 45, 5, "Bien", "Mal", "adios", "hola"]
```

Si no se especifica otra cosa, el orden que se sigue es el de los códigos ascii, por lo que los dígitos numéricos van antes que las letras, y las mayúsculas antes que las minúsculas.

También podemos pasárle una función que le indique cómo ordenar. Esta función debe devolver un valor negativo si el primer elemento es menor, positivo si es mayor, o 0 si son iguales.

Ejemplo: ordenar un array de cadenas sin tener en cuenta si son mayúsculas o minúsculas:

```
let a = ['hola', 'adios', 'Bien', 'Mal']
let b = a.sort(function(elem1, elem2) {
  if (elem1.toLocaleLowerCase() < elem2.toLocaleLowerCase()) return -1;
  if (elem1.toLocaleLowerCase() > elem2.toLocaleLowerCase()) return 1;
  if (elem1.toLocaleLowerCase() = elem2.toLocaleLowerCase()) return 0;
});
// b = ["adios", "Bien", "hola", "Mal"]
```

También se utiliza para ordenar números, tanto de forma ascendente como descendente:

```
let a=[20,4,87,2];
let b= a.sort(function(elem1,elem2){return elem1-elem2}) //ascendente
console.log(b);

b= a.sort(function(elem1,elem2){return elem2 - elem1}) //descendente
console.log(b);
```

Es frecuente utilizar esta función para ordenar arrays de objetos. Por ejemplo, si tenemos un objeto `persona` con los campos `nombre` y `edad`, para ordenar un array de objetos persona por su edad haremos:

```
let persona1 = {nombre: 'Juan', edad: 25};
```

```

let persona2 = {nombre: 'Benito', edad: 52};
let persona3 = {nombre: 'Ana', edad: 33};

let personas = [persona1, persona2, persona3];

let personasOrdenado = personas.sort(function(persona1, persona2) {
    return persona1.edad - persona2.edad
})

```

The screenshot shows a browser developer tools console with the following code and output:

```

let persona1 = {nombre: 'Juan', edad: 25};
let persona2 = {nombre: 'Benito', edad: 52};
let persona3 = {nombre: 'Ana', edad: 33};

let personas = [persona1, persona2, persona3];

personas.sort(function(persona1, persona2) {
    return persona1.edad - persona2.edad
})
console.log(personas);

```

The output window shows:

- No hay problemas
- (3) [{} , {} , {}]
- 0: {nombre: 'Juan', edad: 25}
- 1: {nombre: 'Ana', edad: 33}
- 2: {nombre: 'Benito', edad: 52}
- length: 3
- [[Prototype]]: Array(0)

Usando arrow functions quedaría más sencillo:

```
let personasOrdenado = personas.sort((persona1, persona2) => persona1.edad - persona2.edad)
```

Si lo que queremos es ordenar por un campo de texto podemos usar la función `toLocaleCompare`:

```
let personasOrdenado = personas.sort((persona1, persona2) => persona1.nombre.localeCompare(persona2.nombre))
```

The screenshot shows a browser developer tools console with the following code and output:

```

let persona1 = {nombre: 'Juan', edad: 25};
let persona2 = {nombre: 'Benito', edad: 52};
let persona3 = {nombre: 'Ana', edad: 33};

let personas = [persona1, persona2, persona3];

personas.sort((persona1, persona2) => persona1.nombre.localeCompare(persona2.nombre))
console.log(personas);

```

The output window shows:

- No hay problemas
- (3) [{} , {} , {}]
- 0: {nombre: 'Ana', edad: 33}
- 1: {nombre: 'Benito', edad: 52}
- 2: {nombre: 'Juan', edad: 25}
- length: 3
- [[Prototype]]: Array(0)

EJERCICIO: Haz una función que ordene las notas de un array pasado como parámetro. Si le pasamos [4,8,3,10,5] debe devolver [3,4,5,8,10]. Pruébalo en la consola

## Otros métodos comunes

Otros métodos que se usan a menudo con arrays son:

- `.concat()`: concatena arrays

```

let a = [2, 4, 6]
let b = ['a', 'b', 'c']
let c = a.concat(b)      // c = [2, 4, 6, 'a', 'b', 'c']

```

- `.reverse()`: invierte el orden de los elementos del array

```

let a = [2, 4, 6]
let b = a.reverse()      // b = [6, 4, 2]

```

- `.indexOf()`: devuelve la primera posición del elemento pasado como parámetro o -1 si no se encuentra en el array

- `.lastIndexOf()`: devuelve la última posición del elemento pasado como parámetro o -1 si no se encuentra en el array

```
let a = [2, 4, 6, 4]
console.log(a.indexOf(4));           // 1
console.log(a.lastIndexOf(4));      // 3
console.log(a.lastIndexOf('4'));    // -1
```

## Functional Programming

Se trata de un paradigma de programación (una forma de programar) donde se intenta que el código se centre más en qué debe hacer una función que en cómo debe hacerlo.

El ejemplo más claro es que intenta evitar los bucles `for` y `while` sobre arrays o listas de elementos.

Normalmente, cuando hacemos un bucle es para recorrer la lista y realizar alguna acción con cada uno de sus elementos. Lo que hace *functional programming* es que a la función que debe hacer eso, además de pasarle como parámetro la lista sobre la que debe actuar, se le pasa como segundo parámetro la función que debe aplicarse a cada elemento de la lista.

Desde la versión 5.1 javascript incorpora métodos de *functional programming* en el lenguaje, especialmente para trabajar con arrays:

### filter

Devuelve un nuevo array con los elementos que cumplen determinada condición del array al que se aplica.

Su parámetro es una función, habitualmente anónima, que va interactuando con los elementos del array.

Esta función recibe como primer parámetro el elemento actual del array (sobre el que debe actuar).

Opcionalmente puede tener como segundo parámetro su índice y como tercer parámetro el array completo.

La función debe devolver **true** para los elementos que se incluirán en el array a devolver como resultado, y **false** para el resto.

Ejemplo: dado un array con notas, devolver un array con las notas de los aprobados.

Usando programación *imperativa* (la que se centra en *cómo se deben hacer las cosas*) sería algo como:

```
let arrayNotas = [5.2, 3.9, 6, 9.75, 7.5, 3]
let aprobados = []
for (let i = 0; i < arrayNotas.length; i++) {
  let nota = arrayNotas[i]
  if (nota >= 5) {
    aprobados.push(nota)
  }
} // aprobados = [5.2, 6, 9.75, 7.5]
```

Usando *functional programming* (la que se centra en *qué resultado queremos obtener*) sería:

```
let arrayNotas = [5.2, 3.9, 6, 9.75, 7.5, 3]
let aprobados = arrayNotas.filter(function(nota) {
  if (nota >= 5) {
    return true
  }
})
```

```

} else {
  return false
}
}) // aprobados = [5.2, 6, 9.75, 7.5]

```

Podemos refactorizar esta función para que sea más compacta:

```

let arrayNotas = [5.2, 3.9, 6, 9.75, 7.5, 3]
let aprobados = arrayNotas.filter(function(nota) {
  return nota >= 5 // nota >= 5 se evalúa a 'true' si es cierto, o 'false' si no lo es
})
// aprobados = [5.2, 6, 9.75, 7.5]

```

Y usando funciones tipo flecha la sintaxis queda mucho más simple:

```

let arrayNotas = [5.2, 3.9, 6, 9.75, 7.5, 3]
let aprobados = arrayNotas.filter(nota => nota >= 5)
// aprobados = [5.2, 6, 9.75, 7.5]

```

Las 7 líneas del código usando programación *imperativa* quedan reducidas a sólo una.

EJERCICIO: Dado un array con los días de la semana obtén todos los días que empiezan por 'M'

## find

Como *filter* pero NO devuelve un **array**, devuelve el primer **elemento** que cumpla la condición (o *undefined* si no la cumple ninguno). Ejemplo:

```

let arrayNotas = [4, 5.2, 3.9, 6, 9.75, 7.5, 3]
let primerAprobado = arrayNotas.find(nota => nota >= 5) // primerAprobado = 5.2

```

Este método tiene más sentido con objetos. Por ejemplo, si queremos encontrar un coche de color rojo dentro de un array llamado *coches* cuyos elementos son objetos con un campo 'color', haremos:

```

let coches = [
  {
    "color": "morado",
    "tipo": "berlina",
    "capacidad": 7
  },
  {
    "color": "rojo",
    "tipo": "camioneta",
    "capacidad": 5
  },
  {
    "color": "rojo",
    "tipo": "furgón",
    "capacidad": 7
  }
]

```

```
let cocheBuscado = coches.find(coche => coche.color === 'rojo') // devolverá el objeto completo del primer elemento que cumpla la condición.
```

Si queremos que nos devuelva el objeto en la posición “1” dentro del array, haremos:

```
let cocheBuscado=coches.find((coche,indice) => indice ===1); // devolverá el objeto coches[1]
```

EJERCICIO: Dado un array con los días de la semana obtén el primer día que empieza por ‘M’

## findIndex

Como *find* pero, en vez de devolver el elemento, devuelve su posición (-1 si ningún elemento cumple la condición).

```
let cocheBuscado = coches.findIndex(coche => coche.color === 'rojo') // devolverá 1
```

En el ejemplo de los coches el valor devuelto sería 1, ya que el segundo elemento cumple la condición.

Al igual que el anterior, *findIndex* tiene más sentido con arrays de objetos.

EJERCICIO: Dado un array con los días de la semana, obtén la posición en el array del primer día que empieza por ‘M’.

## every / some

- **every** devuelve **true** si **TODOS** los elementos del array cumplen la condición y **false** en caso contrario.
- **some** devuelve **true** si **ALGÚN** elemento del array cumple la condición. Ejemplo:

```
let arrayNotas = [5.2, 3.9, 6, 9.75, 7.5, 3]
let todosAprobados = arrayNotas.every(nota => nota >= 5) // false
let algunAprobado = arrayNotas.some(nota => nota >= 5) // true
```

EJERCICIO: Dado un array con los días de la semana indica si algún día empieza por ‘S’. Dado un array con los días de la semana indica si todos los días acaban por ‘s’

## map

**map** permite modificar cada elemento de un array y devuelve un nuevo array con los elementos del original modificados.

Ejemplo: queremos subir un 10% cada nota:

```
let arrayNotas = [5.2, 3.9, 6, 9.75, 7.5, 3];
let arrayNotasSubidas = arrayNotas.map(nota => nota + nota * 0.1);
```

EJERCICIO: Dado un array con los días de la semana devuelve otro array con los días en mayúsculas.

## reduce

El método **reduce**:

- Ejecuta una función tipo callback para cada elemento del array, y devuelve **un único valor calculado** a partir de los elementos del array, este valor es el resultado acumulado de las llamadas a la función.
- No se ejecuta la función para elementos vacíos del array.

- No se cambia el contenido del array original.

Sintaxis: `array.reduce(function(total, currentValue, currentIndex, vector), initialValue)`

Hay dos parámetros: una **función** e **initialValue** (este segundo parámetro es opcional).

La función admite 4 parámetros:

- **total o valorAnterior** (obligatorio), contiene el valor devuelto por la función en cada llamada.
- **currentValue** (obligatorio), contiene el valor del elemento actual.
- **currentIndex** (opcional), contiene el índice del elemento actual.
- **vector** (opcional), es el array al que pertenece el elemento actual.

La primera vez que se llama la función, `valorAnterior` y `valorActual` pueden tener uno de dos posibles valores:

- Si se suministró un `valorInicial` al llamar a `reduce`, entonces `valorAnterior` será igual al `valorInicial` y `valorActual` será igual al primer elemento del array.
- Si no se proporcionó un `valorInicial`, entonces `valorAnterior` será igual al primer valor en el array y `valorActual` será el segundo.

Ejemplo: queremos obtener la suma de las notas de un array:

```
let arrayNotas = [4,7,5];
let suma=0;

suma= arrayNotas.reduce((total, valor) => total+= valor, 0);
console.log(suma); // obtiene 16

// Ahora la misma función sin ser tipo flecha
suma= arrayNotas.reduce((total, valor) => {
    return total+= valor
}, 0);
console.log(suma); // obtiene 16
```

La función segunda permite añadir fácilmente otras funcionalidades

Para observar cómo funciona `reduce()`, estudia el siguiente caso para sumar el array de notas. Se ha añadido el parámetro del índice a la función callback.

Si añadimos un `valorInicial`, hace un recorrido por el índice cero. Si este valor es cero, el resultado es el mismo que sin no ponemos `valorInicial`.

<pre>let arrayNotas = [4,7,5]; let suma=0;  // Misma función para sumar con el parámetro índice suma= arrayNotas.reduce((total, valor, i) =&gt; {     console.log('Valor: ' + valor);     console.log('Índice: ' + i);     return total+= valor; }, 0); console.log('Resultado: ' + suma); // obtiene 16</pre>	<p>Niveles predeterminados No hay problemas</p> <table border="1" style="margin-top: 10px; width: 100px;"> <tr><td>Valor: 4</td></tr> <tr><td>Índice: 0</td></tr> <tr><td>Valor: 7</td></tr> <tr><td>Índice: 1</td></tr> <tr><td>Valor: 5</td></tr> <tr><td>Índice: 2</td></tr> <tr><td>Resultado: 16</td></tr> </table>	Valor: 4	Índice: 0	Valor: 7	Índice: 1	Valor: 5	Índice: 2	Resultado: 16	<pre>let arrayNotas = [4,7,5]; let suma=0;  // Misma función para sumar con el parámetro índice suma= arrayNotas.reduce((total, valor, i) =&gt; {     console.log('Valor: ' + valor);     console.log('Índice: ' + i);     return total+= valor; }, ); console.log('Resultado: ' + suma); // obtiene 16</pre>	<p>Filtrar Niveles predeterminado No hay problemas</p> <table border="1" style="margin-top: 10px; width: 100px;"> <tr><td>Valor: 7</td></tr> <tr><td>Índice: 1</td></tr> <tr><td>Valor: 5</td></tr> <tr><td>Índice: 2</td></tr> <tr><td>Resultado: 16</td></tr> </table>	Valor: 7	Índice: 1	Valor: 5	Índice: 2	Resultado: 16
Valor: 4															
Índice: 0															
Valor: 7															
Índice: 1															
Valor: 5															
Índice: 2															
Resultado: 16															
Valor: 7															
Índice: 1															
Valor: 5															
Índice: 2															
Resultado: 16															

En el siguiente ejemplo hay dos casos. Entre el caso de la izda y el de la derecha solo cambia el valor del parámetro opcional `initialValue` que en este caso vale 3, por lo que el resultado cambia.

```

let arrayNotas = [4,7,5];
let suma=0;

// Ahora la misma función sin ser tipo flecha
suma= arrayNotas.reduce((total, valor, i) => {
  console.log('Valor: ' + valor);
  console.log('Índice: ' + i);
  return total+= valor;
},);
console.log('Resultado: ' + suma); // obtiene 16 >
```

Filtrar	Niveles predeterminados
No hay problemas	
Valor: 7 Índice: 1 Valor: 5 Índice: 2 Resultado: 16	

```

let arrayNotas = [4,7,5];
let suma=0;

// Ahora la misma función sin ser tipo flecha
suma= arrayNotas.reduce((total, valor, i) => {
  console.log('Valor: ' + valor);
  console.log('Índice: ' + i);
  return total+= valor;
}, 3); ←
console.log('Resultado: ' + suma); // obtiene 19
```

Filtrar	Niveles predeterminados
No hay problemas	
Valor: 4 Índice: 0 Valor: 7 Índice: 1 Valor: 5 Índice: 2 Resultado: 19	

Otros ejemplos similares de suma de notas:

```

let arrayNotas = [5.2, 3.9, 6, 9.75, 7.5, 3];
let sumaNotas = arrayNotas.reduce((total,nota) => total += nota, 30) // total = 65.35
sumaNotas = arrayNotas.reduce((total,nota) => total += nota) // total = 35.35
```

Podemos tener la función declarada y utilizarla con reduce(). Ejemplo:

```

function suma(a, b) {
  return a + b;
}

const numeros = [10, 20, 30, 40, 50, 60, 70, 80, 90];
const resultado = numeros.reduce(suma);
console.log(resultado); // 450
```

Otro ejemplo: queremos obtener la nota más alta:

```

let arrayNotas = [5.2, 3.9, 6, 9.75, 7.5, 3]
let maxNota = arrayNotas.reduce((max,nota) => nota > max ? nota : max) // max = 9.75
```

Ejemplo muy útil: Integrar un array a partir de varios arrays utilizando el método array.concat()

```

let integrado = [[0,1], [2,3], [4,5]].reduce(function(a,b) {
  return a.concat(b);
});
// integrado es [0, 1, 2, 3, 4, 5]
```

Ejemplo clarificador de la idea: Cadena de montaje, se van añadiendo piezas a la cadena de texto inicial.

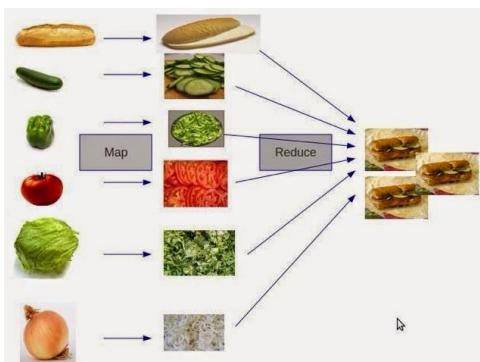
```

const partesDelCoche = ["asientos", "volante", "puertas", "ruedas", "pintura metalizada"];

const coche = partesDelCoche.reduce(function (valorAnterior, valorActual) {
  return `${valorAnterior} ${valorActual},`;
}, "Mi coche tiene: ");

console.log(coche);
//Mi coche tiene: asientos, volante, puertas, ruedas, pintura metalizada,
```

Una idea del funcionamiento de map y reduce sería: Tenemos un “array” de verduras al que le aplicamos una función *map* para que las corte y al resultado le aplicamos un *reduce* para que obtenga un valor (el sandwich) con todas ellas.



EJERCICIO: Dado el array de notas visto anteriormente, usar un método para que devuelva la nota media.

## forEach

Es el método más general de los que hemos visto. No devuelve nada, sino que permite realizar algo con cada elemento del array.

```

let arrayNotas = [5.2, 3.9, 6, 9.75, 7.5, 3]
arrayNotas.forEach((nota, indice, arrayCompleto) => {
  console.log('El elemento de la posición ' + indice + ' es: ' + nota)
})
  
```

Los 3 argumentos de la función son: Valor de cada elemento del array, índice del array, y contenido completo del array.

## includes

Devuelve **true** si el array incluye el elemento pasado como parámetro. Ejemplo:

```

let arrayNotas = [5.2, 3.9, 6, 9.75, 7.5, 3]
arrayNotas.includes(7.5)      // true
  
```

EJERCICIO: Dado un array con los días de la semana indica si algún día es el 'Martes'

## Array.from

Devuelve un array a partir de otro al que se puede aplicar una función de transformación (es similar a *map*). Ejemplo: queremos subir un 10% cada nota:

```

let arrayNotas = [5.2, 3.9, 6, 9.75, 7.5, 3]
let arrayNotasSubidas = Array.from(arrayNotas, nota => nota * 1.1)
  
```

Puede usarse para hacer una copia de un array, igual que *slice*:

```

let arrayA = [5.2, 3.9, 6, 9.75, 7.5, 3];
let arrayB = Array.from(arrayA);
  
```

También se utiliza mucho para convertir colecciones en arrays y así poder usar los métodos de arrays que hemos visto con las colecciones.

Por ejemplo, si queremos mostrar por consola cada párrafo de la página que comience por la palabra 'Si' en primer lugar obtenemos todos los párrafos con:

```
let parrafos = document.getElementsByTagName('p')
```

Esto nos devuelve una colección con todos los párrafos de la página (lo veremos más adelante al ver DOM). Podríamos hacer un **for** para recorrer la colección y mirar los que empiecen por lo indicado, pero no podemos aplicarle los métodos vistos aquí porque son sólo para arrays, así que hacemos:

```
let parrafos = document.getElementsByTagName('p');
let arrayParrafos = Array.from(parrafos);
// y ya podemos usar los métodos que queramos:
arrayParrafos.filter(parrafo => parrafo.textContent.startsWith('Si'))
.forEach(parrafo => alert(parrafo.textContent))
```

**IMPORTANTE: desde este momento se han acabado los bucles *for* en nuestro código para trabajar con arrays. Usaremos siempre estas funciones!!!**

Existen otros métodos para utilizar con arrays, se pueden ver en:

[https://www.w3schools.com/jsref/jsref\\_obj\\_array.asp](https://www.w3schools.com/jsref/jsref_obj_array.asp)

## Referencia vs Copia

Cuando copiamos una variable de tipo *boolean*, *string* o *number* (o se pasa esa variable como parámetro a una función), se hace una copia de la misma. Por lo que, si se modifica, la variable original no es modificada. Ej.:

```
let a = 54
let b = a      // a = 54 b = 54
b = 86        // a = 54 b = 86
```

Sin embargo, al copiar objetos (y los arrays son un tipo de objeto), la nueva variable apunta a la misma posición de memoria que la antigua por lo que los datos de ambas son los mismos:

```
let a = [54, 23, 12]
let b = a      // a = [54, 23, 12] b = [54, 23, 12]
b[0] = 3      // a = [3, 23, 12] b = [3, 23, 12]
let fecha1 = new Date('2022-09-23')
let fecha2 = fecha1      // fecha1 = '2022-09-23'    fecha2 = '2022-09-23'
fecha2.setFullYear(1999) // fecha1 = '1999-09-23'    fecha2 = '1999-09-23'
```

Si queremos obtener una copia de un array que sea independiente del original podemos obtenerla con *slice* o con *Array.from*:

```
let a = [2, 4, 6]
let copiaDeA = a.slice()  // ahora ambos arrays contienen lo mismo, pero son diferentes
let otraCopiaDeA = Array.from(a)
```

En el caso de objetos, es algo más complejo. ES6 incluye **Object.assign**, que hace una copia de un objeto:

```
let a = {id:2, name: 'object 2'}
let copiaDeA = Object.assign({}, a) //ahora ambos objetos contienen lo mismo, pero son diferentes
```

Sin embargo, si el objeto tiene como propiedades otros objetos, éstos se continúan pasando por referencia. En ese caso lo más sencillo sería utilizar *objetoCopia = JSON.stringify(objetoOriginal);*

**JSON** es un formato de texto para almacenar y transportar datos.

JavaScript tiene una función integrada para convertir cadenas JSON en objetos JavaScript: `JSON.parse()`

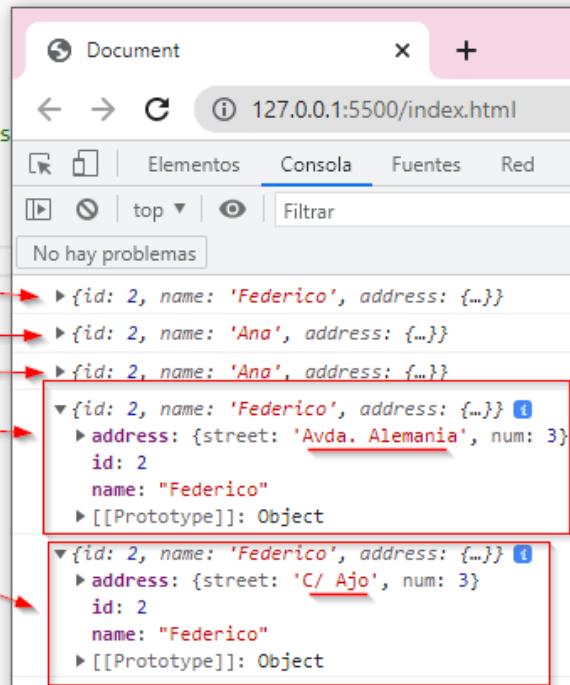
JavaScript tiene una función integrada para convertir un objeto en una cadena JSON: `JSON.stringify()`

```
let a = {id: 2, name: 'object 2', address: {street: 'C/ Ajo', num: 3} }
let copiaDeA = JSON.parse(JSON.stringify(a)) // ahora ambos objetos contienen lo mismo pero
son diferentes
```

```
'use strict';
let a = {id: 2, name: 'Federico', address: {street: 'C/ Ajo', num: 3} }
let b = a;
let copia1= Object.assign({}, a);

//solución:
// ahora ambos objetos contienen lo mismo pero son diferentes
let copiaDeA = JSON.parse(JSON.stringify(a))

console.log((b));
a.name= 'Ana';
a.address.street = 'Avda. Alemania'
console.log(a);
console.log(b);
console.log(copia1);
console.log(copiaDeA);
```



EJERCICIO: Dado el array arr1 con los días de la semana haz un array arr2 que sea igual al arr1. Elimina de arr2 el último día y comprueba qué ha pasado con arr1. Repita la operación con un array llamado arr3 pero que crearás haciendo una copia de arr1.

## Rest y Spread

También podemos copiar objetos usando *rest* y *spread*.

Permiten extraer a parámetros los elementos de un array o string (*spread*) o convertir en un array un grupo de parámetros (*rest*). El operador de ambos es ... (3 puntos).

**Rest** se utiliza para convertir en un array un grupo de parámetros (*rest*). El operador es ... (3 puntos).

Para usar *rest* como parámetro de una función debe ser siempre el último parámetro:

Ejemplo: queremos hacer una función que calcule la media de las notas que se le pasen como parámetro y que no sabemos cuántas son. Para llamar a la función haremos:

```
console.log(notaMedia(3.6, 6.8))
console.log(notaMedia(5.2, 3.9, 6, 9.75, 7.5, 3))
```

La función notaMedia convertirá los parámetros recibidos en un array usando *rest*:

```
function notaMedia(...notas) {
```

```
let total = notas.reduce((total,nota) => total += nota)
return total/notas.length
}
```

**Spread** permite pasar como parámetros independientes los elementos de un array o string. El operador también es ... (3 puntos).

Si lo que queremos es convertir un array en un grupo de elementos haremos *spread*.

```
let array = [1,2,3];
console.log(array);
// hacemos la conversión de array a parámetros
console.log(...array);
```

Por ejemplo, el objeto *Math* proporciona métodos para trabajar con números como *.max* que devuelve el máximo de los números pasados como parámetro. Para saber la nota máxima en vez de utilizar *.reduce* como hicimos en el ejemplo anterior, podemos hacer:

```
let arrayNotas = [5.2, 3.9, 6, 9.75, 7.5, 3];
let maximo = Math.max(arrayNotas); //maximo = NaN
// si hacemos Math.max(arrayNotas) devuelve NaN porque arrayNotas es un array y no un número

// hacemos la conversión de array a parámetros
let maximoSpread = Math.max(...arrayNotas); // maxNota = 9.75
```

Estas funcionalidades nos ofrecen otra manera de copiar objetos (pero sólo a partir de ES-2018):

```
let a = {id: 2, name: 'object 2'}
let copiaDeA = { ...a} // ahora ambos objetos contienen lo mismo pero son diferentes
```

```
let b = [2, 8, 4, 6]
let copiaDeB = [ ...b ] // ahora ambos objetos contienen lo mismo pero son diferentes
```

## Desestructuración de arrays

Similar a *rest* y *spread*, permiten extraer los elementos del array directamente a variables y viceversa. Ejemplo:

```
let arrayNotas = [5.2, 3.9, 6, 9.75, 7.5, 3]
let [primera, segunda, tercera] = arrayNotas // primera = 5.2, segunda = 3.9, tercera = 6
let [primera, , , cuarta] = arrayNotas // primera = 5.2, cuarta = 9.75
let [primera, ...resto] = arrayNotas // primera = 5.2, resto = [3.9, 6, 9.75, 3]
```

También se pueden asignar valores por defecto:

```
let preferencias = ['Javascript', 'NodeJS']
let [lenguaje, backend = 'Laravel', frontend = 'VueJS'] = preferencias
    // lenguaje = 'Javascript', backend = 'NodeJS', frontend = 'VueJS'
```

La desestructuración también funciona con objetos. Es normal pasar un objeto como parámetro para una función, pero si sólo nos interesan algunas propiedades del mismo, podemos desestructurarlo:

```
const miProducto = {
  id: 5,
  name: 'TV Samsung',
  units: 3,
  price: 395.95
};

// Se puede abreviar: function muestraNombre({name, units}) {
function muestraNombre({name: name, units: units}) {
  console.log('Del producto ' + name + ' hay ' + units + ' unidades')
}

muestraNombre(miProducto); //Del producto TV Samsung hay 3 unidades
```

También podemos asignar valores por defecto:

```
function muestraNombre({name, units = 0}) {
  console.log('Del producto ' + name + ' hay ' + units + ' unidades')
}

muestraNombre({name: 'USB Kingston'});
// mostraría: Del producto USB Kingston hay 0 unidades
```

## Los objetos Map y Set

En el ámbito de JavaScript, a menudo los desarrolladores pasan mucho tiempo decidiendo la estructura de datos correcta que se usará. Esto se debe a que elegir la estructura de datos correcta puede facilitar la manipulación de esos datos posteriormente, con lo cual se puede ahorrar tiempo y facilitar la comprensión del código.

Las dos estructuras de datos predominantes para almacenar conjuntos de datos son los Object (Objetos) y Array (un tipo de objeto).

Los desarrolladores utilizan Object para almacenar pares clave-valor y Array para almacenar listas indexadas.

Sin embargo, para dar más flexibilidad a los desarrolladores, en la especificación ECMAScript 2015 se introdujeron dos nuevos tipos de objetos iterables:

- los Map, que son grupos ordenados de pares clave-valor.
- los Set, que son grupos de valores únicos.

### Map

El objeto **Map** es una colección de parejas de [clave,valor].

En cambio, un objeto en Javascript es un tipo particular de *Map* en el que las claves sólo pueden ser texto o números.

Se puede acceder a una propiedad con `.` o **[propiedad]**. Ejemplo:

```
let persona = {
  nombre: 'John',
  apellido: 'Doe',
```

```

    edad: 39
}
console.log(persona.nombre)      // John
console.log(persona['nombre'])   // John

```

Un *Map* permite que la clave sea cualquier cosa (array, objeto, ...).

Más información en [MDN](#) o cualquier otra página.

## Set

El objeto **Set** es como un *Map*, pero que no almacena los valores, almacena solo la clave. Podemos verlo como una colección que no permite duplicados.

Tiene la propiedad **size** que devuelve su tamaño y los métodos **.add** (añade un elemento), **.delete** (lo elimina) o **.has** (indica si el elemento pasado se encuentra o no en la colección) y también podemos recorrerlo con **.forEach**.

Una forma sencilla de eliminar los duplicados de un array es crear con él un *Set*:

```

let ganadores = ['Márquez', 'Rossi', 'Márquez', 'Lorenzo', 'Rossi', 'Márquez',
'Márquez']
let ganadoresNoDuplicados = new Set(ganadores)      // {'Márquez', 'Rossi', 'Lorenzo'}

// o si lo queremos en un array:

ganadoresNoDuplicados = Array.from(new Set(ganadores)) // ['Márquez', 'Rossi', 'Lorenzo']

```



## DWEC – Javascript Web Cliente.

JavaScript 03 – Objetos y Clases en Javascript.....	1
Introducción.....	1
Propiedades de un objeto.....	2
Agregar y eliminar propiedades en un objeto:.....	4
Métodos get y set en objetos .....	4
Clases en Javascript.....	4
La palabra reservada this. Cuidado. ....	5
Herencia, sobreescritura, polimorfismo.....	6
Métodos estáticos.....	8
Atributos estáticos .....	9
Constantes estáticas .....	10
toString() .....	11
valueOf() .....	14
Métodos get y set en JS .....	14
Agregar propiedades y métodos a los objetos de una clase (prototype).....	15
Prototipos y POO en JS5.....	16
Uso de call y apply en Javascript.....	16
La clase Object .....	17

# JavaScript 03 – Objetos y Clases en Javascript

## Introducción

En Javascript podemos definir cualquier variable como un objeto, existen dos formas para hacerlo:

- Declarándola con **new** (NO se recomienda)
- Forma recomendada: creando un *literal object* usando notación **JSON**.

Ejemplo con *new*:

```
let alumno = new Object();
alumno.nombre = 'Carlos';      // se crea la propiedad 'nombre' y se le asigna un valor
alumno['apellidos'] = 'Pérez Ortiz';    // se crea la propiedad 'apellidos'
alumno.edad = 19;
```

Creando un *literal object* según la forma recomendada, el ejemplo anterior sería:

```
let alumno = {
  nombre: 'Carlos',
  apellidos: 'Pérez Ortiz',
```

```
edad: 19,  
};
```

## Propiedades de un objeto

Podemos acceder a las propiedades con . (punto) o [ ]:

```
console.log(alumno.nombre);           // imprime 'Carlos'  
console.log(alumno['nombre']);       // imprime 'Carlos'  
  
let prop = 'nombre';  
console.log(alumno[prop]);          // imprime 'Carlos'
```

Si intentamos acceder a propiedades que no existen no se produce un error, se devuelve *undefined*:

```
console.log(alumno.ciclo);          // muestra undefined
```

Sin embargo, se genera un error si intentamos acceder a propiedades de algo que no es un objeto:

```
console.log(alumno.ciclo);          // muestra undefined  
console.log(alumno.ciclo.descrip);  // se genera un ERROR
```

En versiones anteriores de JavaScript, para evitar este error se comprobaba que existían las propiedades previamente. Veamos un ejemplo:

```
console.log(alumno.ciclo && alumno.ciclo.descrip);  
// si alumno.ciclo es un objeto muestra el valor de  
// alumno.ciclo.descrip y si no muestra undefined
```

Con ES2020 (ES11) se ha incluido el operador ?. para evitar tener que comprobar esto nosotros:

```
console.log(alumno.ciclo?.descrip);  
// si alumno.ciclo es un objeto muestra el valor de  
// alumno.ciclo.descrip y si no muestra undefined
```

Este nuevo operador también puede aplicarse a **arrays**:

```
let alumnos = ['Juan', 'Ana'];  
console.log(alumnos?.[0]);  
// si alumnos es un array y existe el primer elemento muestra el valor  
// si ese elemento no existe muestra undefined  
// si no existe el objeto con el nombre alumnos da ERROR
```

Podremos recorrer las propiedades de un objeto con `for..in`:

```
for (let prop in alumno) {  
  console.log(prop + ': ' + alumno[prop])  
}
```

Resultado:

```
for (let prop in alumno) {
    console.log(prop + ': ' + alumno[prop])
}
```

nombre: Carlos
apellidos: Pérez Ortiz
edad: 19

Una propiedad de un objeto puede ser una función:

```
alumno.getInfo = function() {
    return 'El alumno ' + this.nombre + ' ' + this.apellidos + ' tiene ' + this.edad +
    ' años'
}

console.log(alumno.getInfo()); //El alumno Carlos Pérez Ortiz tiene 19 años
```

También se puede incluir la declaración del método en la declaración del objeto:

```
let alumno = {
    nombre: 'Carlos',
    apellidos: 'Pérez Ortiz',
    edad: 19,
    getInfo: function(){
        return 'El alumno ${this.nombre} ${this.apellidos} tiene ${this.edad} años';
    }
};

console.log(alumno.getInfo()); //El alumno Carlos Pérez Ortiz tiene 19 años
```

OJO: deberíamos poder ponerlo con sintaxis *arrow function*, pero no funciona.

```
let alumno = {
    nombre: 'Carlos',
    apellidos: 'Pérez Ortiz',
    edad: 19,
    getInfo: ()=> `El alumno ${this.nombre} ${this.apellidos} tiene ${this.edad} años`
};

console.log(alumno.getInfo()); //El alumno undefined undefined tiene undefined años
```

No funciona bien porque *this* tiene distinto valor dependiendo del contexto, y no se puede usar en estos casos con función flecha. Tienes un documento titulado “JavaScript - Anexo - Uso de *this* en contexto” que lo explica.

Si el valor de una propiedad es el valor de una variable que se llama como ella, desde ES2015 no es necesario ponerlo:

```
let nombre = 'Carlos'
let alumno = {
    nombre,           // es equivalente a nombre: nombre
    apellidos: 'Pérez Ortiz',
    ...
}
```

EJERCICIO: Crea un objeto llamado tvSamsung con las propiedades **nombre** (TV Samsung 42’), **categoria** (Televisores), **unidades** (4), **precio** (345.95) y con un método llamado **importe** que devuelve el valor total de las unidades (**nº de unidades \* precio**).

Prueba el uso del método con un ejemplo.

### Agregar y eliminar propiedades en un objeto:

Se pueden agregar propiedades sin más. Pero OJO!! Si nos equivocamos al modificar el valor de una propiedad (escribimos un nombre de propiedad distinto) nos va a crear una propiedad que no deseamos.

Para eliminar una propiedad se hace con la palabra reservada **delete**

```
let alumno = {
  nombre: 'Carlos',
  apellidos: 'Pérez Ortiz',
  edad: 19,
};

//añadimos la propiedad email
alumno.email='cperez@email.com';
console.log(alumno.email); //cperez@email.com

//eliminamos la proiedad email
delete alumno.email;
console.log(alumno); // {nombre: 'Carlos', apellidos: 'Pérez Ortiz', edad: 19}
```

### Métodos get y set en objetos

Al igual que en las clases (como veremos más adelante), se pueden declarar métodos get y set en una o en varias propiedades de un objeto para acceder a su contenido o para modificar su valor.

## Clases en Javascript

Desde ES2015 la POO en Javascript es similar a como se hace en otros lenguajes: clases, herencia, cohesión, abstracción, polimorfismo, acoplamiento, encapsulamiento... En Javascript solo se permite un método constructor.

Más información en [https://es.wikipedia.org/wiki/Programaci%C3%B3n\\_orientada\\_a\\_objetos](https://es.wikipedia.org/wiki/Programaci%C3%B3n_orientada_a_objetos)

Veamos un ejemplo:

```
class Alumno {
  constructor(nombre, apellidos, edad) {
    this.nombre = nombre
    this.apellidos = apellidos
    this.edad = edad
  }
  getInfo() {
    return 'El alumno ' + this.nombre + ' ' + this.apellidos + ' tiene ' + this.edad + ' años'
  }
}

let alumno1 = new Alumno('Carlos', 'Pérez Ortiz', 19)
console.log(alumno1.getInfo()) // imprime 'El alumno Carlos Pérez Ortiz tiene 19 años'
```

Recuerda que para las funciones y variables tipo var, da igual el orden (declaración primero y llamada después), o viceversa, pero para las clases no. Primero se declara la clase y luego se crean objetos.

Para las clases no se aplica el concepto de *hoisting*, por lo que no es posible crear objetos antes de declarar la clase.

**EJERCICIO:** Crea una clase Productos con las propiedades y métodos del ejercicio anterior (el de la TV). Además tendrá un método **getInfo** que devolverá: ‘Nombre (categoría): unidades uds x precio € = importe €’. Crea 3 productos diferentes y prueba getInfo.

## La palabra reservada this. Cuidado.

Dentro de una función se crea un nuevo contexto y la variable *this* pasa a hacer referencia a dicho contexto. Si en el ejemplo anterior hiciéramos algo como esto:

```
class Alumno {
    constructor(nombre, apellidos, edad) {
        this.nombre = nombre;
        this.apellidos = apellidos;
        this.edad = edad;
    }
    getInfo() {
        function nomAlum() {
            return this.nombre + ' ' + this.apellidos; // Aquí this no es el objeto Alumno
        }
        return 'El alumno' + nomAlum() + ' tiene ' + this.edad + ' años';
    }
}
```

Este código fallaría porque dentro de *nomAlum* la variable *this* ya no hace referencia al objeto *Alumno* sino al contexto de la función. Este ejemplo no tiene mucho sentido, pero a veces nos pasará en manejadores de eventos.

Si debemos llamar a una función dentro de un método (o de un manejador de eventos) tenemos varias formas de pasarle el valor de *this*:

**1<sup>a</sup> forma:** Usando una *arrow function* que no crea un nuevo contexto, por lo que *this* conserva su valor.

```
getInfo() {
    let nomAlum=() => this.nombre + ' ' + this.apellidos;
    return 'El alumno ' + nomAlum() + ' tiene ' + this.edad + ' años';
}
```

**2<sup>a</sup> forma:** Pasándole *this* como parámetro a la función:

```
getInfo() {
    function nomAlum(alumno){
        return alumno.nombre + ' ' + alumno.apellidos;
    }
    return 'El alumno ' + nomAlum(this) + ' tiene ' + this.edad + ' años';
}
```

**3<sup>a</sup> forma:** Guardando el valor de *this* en otra variable (como *that*)

```
getInfo() {
    let that=this;
```

```

function nomAlum() {
    return that.nombre + " " + that.apellidos;
}
return "El alumno " + nomAlum() + " tiene " + this.edad + " años";
}

```

**4<sup>a</sup> forma:** Haciendo un *bind* de *this*, se explica en el apartado de eventos.

## Herencia, sobreescritura, polimorfismo

**Nota:** Las relaciones entre las clases se dan por **Herencia** (Especialización o Generalización), **Asociación**, **Agregación** (por Referencia), **Composición** (por Valor) y **Dependencia** (Uso). Más información en Anexo - Introducción a UML en el apartado “2.5 Diagrama de Clases”.

Una clase puede heredar de otra utilizando la palabra reservada **extends** y heredará todas sus propiedades y métodos.

Los métodos podemos sobrescribirlos en la clase hija (seguimos pudiendo llamar a los métodos de la clase padre utilizando la palabra reservada **super** (que es lo que haremos si creamos un constructor en la clase hija)).

```

class Alumno {
    constructor(nombre, apellidos, edad) {
        this.nombre = nombre;
        this.apellidos = apellidos;
        this.edad = edad;
    }
    getInfo() {
        return (
            "El alumno " + this.nombre + " " + this.apellidos + " tiene " + this.edad + " años"
        );
    }
}

class AlumnInf extends Alumno {
    constructor(nombre, apellidos, edad, ciclo) {
        super(nombre, apellidos, edad);
        this.ciclo = ciclo;
    }
    getGradoMedio() {
        if (this.ciclo.toUpperCase() === "SMR") return true;
        else return false;
    }
    getInfo() {
        return (
            super.getInfo() + " y estudia el Grado " +
            (this.getGradoMedio() ? "Medio" : "Superior") + " de " + this.ciclo );
    }
}

let azp= new Alumno("Ana", "Zubiri Peláez", 24);
console.log(azp.getInfo()); // imprime: 'El alumno Ana Zubiri Peláez tiene 24 años'

let cpo = new AlumnInf("Carlos", "Pérez Ortiz", 19, "DAW");

```

```
console.log(cpo.getInfo()); // imprime 'El alumno Carlos Pérez Ortiz tiene 19 años y
estudia el Grado Superior de DAW'
```

En el ejemplo anterior hemos sobreescrito el método `getInfo()`, lo que implica que hay dos versiones de este método (la primera en la clase padre y la segunda en la clase hija). Se aplicará la primera versión de `getInfo()` o la segunda, dependiendo de si el objeto en cuestión es de una clase o de otra.

Siguiendo con el ejemplo visto anteriormente:

- Quitamos la salida por consola de ambos objetos
- Declaramos una función **imprimir()** fuera de las dos clases.
- Llamamos a la función imprimir pasando como argumento cada objeto

Al llamar a imprimir pasando como argumento cada objeto (cada uno es de un tipo), conseguimos que se llame, a su vez, al método `getInfo()` de la clase de ese objeto.

```
class Alumno {
  constructor(nombre, apellidos, edad) {
    this.nombre = nombre;
    this.apellidos = apellidos;
    this.edad = edad;
  }
  getInfo() {
    return (
      "El alumno " + this.nombre + " " + this.apellidos + " tiene " + this.edad + " años"
    );
  }
}

class AlumnInf extends Alumno {
  constructor(nombre, apellidos, edad, ciclo) {
    super(nombre, apellidos, edad);
    this.ciclo = ciclo;
  }
  getGradoMedio() {
    if (this.ciclo.toUpperCase() === "SMR") return true;
    else return false;
  }
  getInfo() {
    return (
      super.getInfo() + " y estudia el Grado " +
      (this.getGradoMedio() ? "Medio" : "Superior") + " de " + this.ciclo );
  }
}

let azp= new Alumno("Ana", "Zubiri Peláez", 24);
let cpo = new AlumnInf("Carlos", "Pérez Ortiz", 19, "DAW");

function imprimir(objeto){
  console.log(objeto.getInfo());
}
```

```
imprimir(azp);
imprimir(cpo);
```

Es lo que se llama **polimorfismo**, pues hay múltiples formas en tiempo de ejecución, y se ejecutará un método u otro dependiendo de la referencia pasada como argumento.

### *InstanceOf*

Para saber el tipo con el que trabajamos se utiliza **instanceof**

Así sabremos el tipo de objeto que estamos recibiendo.

```
function imprimir(objeto){
    console.log(objeto.getInfo());
    if (objeto instanceof AlumnInf){
        console.log(objeto.nombre + ' es un alumno de informática');
    }
}
```

En el ejemplo anterior, preguntamos si el objeto pasado como argumento es un tipo de la clase **AlumnInf**, de ser así, muestra por consola la frase “Fulanito es un alumno de informática”.

Hay que tener en cuenta que los objetos de la clase **AlumnInf** son también de la clase **Alumno** y también de la clase **Object**. Así que, para preguntar por el tipo de un parámetro, se suele hacer empezando por el tipo de menor jerarquía (el hijo).

EJERCICIO: crea una clase Televisores que hereda de Productos y que tiene una nueva propiedad llamada **tamaño**. El método **getInfo** mostrará el tamaño junto al nombre.

### Métodos estáticos

Desde ES2015 podemos declarar métodos estáticos, pero no propiedades estáticas. Estos métodos se llaman directamente **utilizando el nombre de la clase** y no tienen acceso al objeto **this** (ya que no hay objeto instanciado).

```
class User {
    static getRoles() {
        return ["user", "guest", "admin"]
    }
}

console.log(User.getRoles()) // ["user", "guest", "admin"]
let usuario = new User("john")
console.log(usuario.getRoles()) // Uncaught TypeError: usuario.getRoles is not a function
```

El siguiente ejemplo demuestra varias cosas:

1. Cómo se implementa método estático en una clase,
2. Que una clase con un miembro estático puede ser sub-claseada.
3. Finalmente demuestra cómo un método estático puede (y cómo no) ser llamado.

```
class Tripple {
    static tripple(n) {
        n = n || 1;
        return n * 3;
```

```

        }
    }

    class BiggerTripple extends Tripple {
        static tripple(n) {
            return super.tripple(n) * super.tripple(n);
        }
    }

    console.log(Tripple.tripple());           // 3
    console.log(Tripple.tripple(3));          // 9
    console.log(BiggerTripple.tripple(3));    // 81
    let tp = new Tripple();
    console.log(tp.tripple()); //ERROR Logs 'tp.tripple is not a function'.

```

## Atributos estáticos

Se pueden definir atributos estáticos con la palabra reservada **static**, y será un atributo que pertenece a la clase y no al objeto.

Para hacer referencia (utilizar este atributo) hay que poner *NombreDeClase.atributo*

Es muy útil para llevar un contador de objetos de la clase, o para llevar un atributo de clave única autoincremenetal (que en nuestro caso hemos llamado *id*).

```

class Alumno {
    static contadorObjetosAlumno = 0;
    constructor(nombre, apellidos, edad) {
        this.id = ++ Alumno.contadorObjetosAlumno; // cada vez que se crea un objeto,
                                                    // se incrementa
        this.nombre = nombre;
        this.apellidos = apellidos;
        this.edad = edad;
        console.log(this);
    }
}

let misAlumnos=[];
misAlumnos[0] = new Alumno('Carlos', 'Pérez Ortiz', 19);
misAlumnos[1] = new Alumno('Rosa', 'Cifuentes Martín', 24);
misAlumnos.push(new Alumno('Ana', 'Zubiri Peláez', 29));

```

▶ Alumno {id: 1, nombre: 'Carlos', apellidos: 'Pérez Ortiz', edad: 19}
▶ Alumno {id: 2, nombre: 'Rosa', apellidos: 'Cifuentes Martín', edad: 24}
▶ Alumno {id: 3, nombre: 'Ana', apellidos: 'Zubiri Peláez', edad: 29}

Este atributo contador también se hereda a las clases hijas de Alumno si las hubiere. Cada vez que se crea un objeto de una clase hija, incrementa este contador.

Resumiendo: Hemos creado una variable que tiene un identificador único para cada objeto que se crea, utilizando una variable estática.

Debido a que la variable estática se asocia con la clase y no con los objetos, permite asignar un valor único. Que en el ejemplo va preincrementándose para lograr que el id sea único y consecutivo, independientemente de que sea un objeto de la clase padre (donde está definido) o en la clase que hereda.

## Constantes estáticas utilizando métodos estáticos

Para disponer de una variable estática de solo lectura, no podemos utilizar la palabra reservada **const**.

Lo que haremos es crear un método estático que solo permitirá leer el valor que devuelve el método, sin poder modificar este valor. Así parecerá que es una constante estática.

En el ejemplo siguiente declararemos la constante MAX\_OBJ con el valor 5. (el máximo número de objetos que podremos crear serán 5, tanto de la clase padre como de la hija).

```
class Alumno {
    static contadorObjetosAlumno = 0;
    static get MAX_OBJ(){
        return 5;
    }
    constructor(nombre, apellidos, edad) {
        this.id = ++ Alumno.contadorObjetosAlumno; // cada vez que se crea un objeto,
                                                    // se incrementa
        this.nombre = nombre;
        this.apellidos = apellidos;
        this.edad = edad;
        console.log(this);
    }
}

console.log(Alumno.MAX_OBJ); // 5
Alumno.MAX_OBJ = 10;
console.log(Alumno.MAX_OBJ); // 5
```

Como hemos definido un método **get**, podemos recuperar esta información de la forma **Alumno.MAX\_OBJ**

Como no hemos definido método **set** para este atributo, no podremos modificar el valor.

Aunque no hemos creado una variable, este método funciona como si fuera una variable (constante).

Funciona igual que un atributo de la clase.

Siguiendo con el ejemplo: cambiamos el constructor:

```
class Alumno {
    static contadorObjetosAlumno = 0;
    static get MAX_OBJ(){
        return 5;
    }
    constructor(nombre, apellidos, edad) {
        if(Alumno.contadorObjetosAlumno < Alumno.MAX_OBJ){
            this.id = ++ Alumno.contadorObjetosAlumno;
        } else {
```

```

        console.log('Se ha superado el máximo de objetos permitidos de la clase
Alumno');
    }
    this.nombre = nombre;
    this.apellidos = apellidos;
    this.edad = edad;
    console.log(this);
}
}

let misAlumnos=[];
misAlumnos[0] = new Alumno('Carlos', 'Pérez Ortiz', 19);
misAlumnos[1] = new Alumno('Rosa', 'Cifuentes Martín', 24);
misAlumnos[2] = new Alumno('Ana', 'Zubiri Peláez', 29);
misAlumnos[3] = new Alumno('Rosa', 'Cifuentes Martín', 24);
misAlumnos[4] = new Alumno('Ana', 'Zubiri Peláez', 29);
misAlumnos[5] = new Alumno('Ana', 'Zubiri Peláez', 29);

```

Importante a tener en cuenta: En el caso anterior se crearán 6 alumnos, el sexto alumno se ha creado porque no hemos mandado llamar a una excepción, pero no tiene atributo id porque así lo hemos decidido en el constructor:

```

▶ Alumno {id: 1, nombre: 'Carlos', apellidos: 'Pérez Ortiz', edad: 19}
▶ Alumno {id: 2, nombre: 'Rosa', apellidos: 'Cifuentes Martín', edad: 24}
▶ Alumno {id: 3, nombre: 'Ana', apellidos: 'Zubiri Peláez', edad: 29}
▶ Alumno {id: 4, nombre: 'Rosa', apellidos: 'Cifuentes Martín', edad: 24}
▶ Alumno {id: 5, nombre: 'Ana', apellidos: 'Zubiri Peláez', edad: 29}
Se ha superado el máximo de objetos permitidos de la clase Alumno
▶ Alumno {nombre: 'Ana', apellidos: 'Zubiri Peláez', edad: 29}
▶ misAlumnos
▶ ▶ (6) [Alumno, Alumno, Alumno, Alumno, Alumno, Alumno]

```

## toString()

De forma implícita, todas las clases que vayamos a crear heredan (*extends*) de la clase *Object*. En Javascript, la clase *Object* es la clase padre de todas las clases, por tanto, podemos usar el método heredado *toString()* o sobrescribirlo.

Al convertir un objeto a string (por ejemplo, al concatenarlo con un String) se llama al método ***.toString()*** del propio objeto, que por defecto devuelve la cadena [object Object]. Podemos sobrecargar este método para que devuelva lo que queramos:

```

class Alumno {
  constructor(nombre, apellidos, edad) {
    this.nombre = nombre;
    this.apellidos = apellidos;
    this.edad = edad;
  }

  toString() {

```

```

        return this.apellidos + ', ' + this.nombre
    }
}

let cpo = new Alumno('Carlos', 'Pérez Ortiz', 19);
console.log('Alumno:' + cpo)      // imprime 'Alumno: Pérez Ortiz, Carlos'
                                  // en vez de 'Alumno: [object Object]'
```

Este método también es el que se usará si queremos ordenar una array de objetos (recuerda que `.sort()` ordena alfabéticamente para lo que llama al método `.toString()` del objeto a ordenar).

Por ejemplo, tenemos el array de alumnos `misAlumnos` que queremos ordenar alfabéticamente por apellidos. Si la clase `Alumno` no tiene un método `toString` habría que hacer como vimos en el tema de Arrays:

```

let misAlumnos=[];
misAlumnos[0] = new Alumno('Carlos', 'Pérez Ortiz', 19);
misAlumnos[1] = new Alumno('Rosa', 'Cifuentes Martín', 24);
misAlumnos[2] = new Alumno('Ana', 'Zubiri Peláez', 29);
misAlumnos.sort(function(alum1, alum2){
    if (alum1.apellidos > alum2.apellidos) return 1
    if (alum1.apellidos < alum2.apellidos) return -1
});
```

**NOTA:** como las cadenas a comparar pueden tener acentos u otros caracteres propios del idioma, el código anterior no siempre funcionará bien. La forma correcta de comparar cadenas es usando el método `.localeCompare()`. El código anterior debería ser:

```

misAlumnos.sort(function(alum1, alum2) {
    return alum1.apellidos.localeCompare(alum2.apellidos)
});
```

que con *arrow function* quedaría:

```
misAlumnos.sort((alum1, alum2) => alum1.apellidos.localeCompare(alum2.apellidos) )
```

o si queremos comparar por 2 campos ('apellidos' y 'nombre')

```

misAlumnos.sort((alum1, alum2) =>
(alum1.apellidos+alum1.nombre).localeCompare(alum2.apellidos+alum2.nombre) )
```

Si sobreescrivimos el método `toString`, podemos utilizar este método para la ordenación:

```

class Alumno {
constructor(nombre, apellidos, edad) {
    this.nombre = nombre;
    this.apellidos = apellidos;
    this.edad = edad;
}

toString() {
    return this.apellidos + ', ' + this.nombre
}
}
```

```

let misAlumnos=[];
misAlumnos[0] = new Alumno('Carlos', 'Pérez Ortiz', 19);
misAlumnos[1] = new Alumno('Rosa', 'Cifuentes Martín', 24);
misAlumnos[2] = new Alumno('Ana', 'Zubiri Peláez', 29);
misAlumnos.sort(function(alum1, alum2){
  if (alum1.toString() > alum2.toString()) return 1;
  if (alum1.toString() < alum2.toString()) return -1;
});

```

Pero con el método `toString` que hemos definido antes podemos hacer directamente:

```
misAlumnos.sort()
```

Quedando el ejemplo al completo:

```

class Alumno {
  constructor(nombre, apellidos, edad) {
    this.nombre = nombre;
    this.apellidos = apellidos;
    this.edad = edad;
  }

  toString() {
    return this.apellidos + ', ' + this.nombre
  }
}

let misAlumnos=[];
misAlumnos[0] = new Alumno('Carlos', 'Pérez Ortiz', 19);
misAlumnos[1] = new Alumno('Rosa', 'Cifuentes Martín', 24);
misAlumnos[2] = new Alumno('Ana', 'Zubiri Peláez', 29);
misAlumnos.sort();

```

Obteniendo en la consola:

```

misAlumnos
▼ (3) [Alumno, Alumno, Alumno] ⓘ
  ► 0: Alumno {nombre: 'Rosa', apellidos: 'Cifuentes Martín', edad: 24}
  ► 1: Alumno {nombre: 'Carlos', apellidos: 'Pérez Ortiz', edad: 19}
  ► 2: Alumno {nombre: 'Ana', apellidos: 'Zubiri Peláez', edad: 29}
    length: 3
  ► [[Prototype]]: Array(0)

```

**NOTA:** si queremos ordenar un array de objetos por un campo numérico lo más sencillo es restar dicho campo:

```
misAlumnos.sort((alum1, alum2) => alum1.edad - alum2.edad)
```

EJERCICIO: modifica las clases Productos y Televisores para que el método que muestra los datos del producto se llame de una manera más adecuada.

EJERCICIO: Crea 5 productos y guárdalos en un array. Crea las siguientes funciones (todas reciben ese array como parámetro):

prodsSortByName: devuelve un array con los productos ordenados alfabéticamente.

prodsSortByPrice: devuelve un array con los productos ordenados por precio.

prodsTotalPrice: devuelve el importe total de los productos del array, con 2 decimales.

prodsWithLowUnits: además del array recibe como segundo parámetro un número, y devuelve un array con todos los productos de los que quedan menos de las unidades indicadas.

prodsList: devuelve una cadena que dice ‘Listado de productos:’ y en cada línea un guión y la información de un producto del array.

## valueOf()

Al comparar objetos (con >, <, ...) se usa el valor devuelto por el método `.toString()`

Pero si definimos un método `.valueOf()` será este el que se usará en comparaciones:

```
class Alumno {
  ...
  valueOf() {
    return this.edad
  }
}

let cpo = new Alumno('Carlos', 'Pérez Ortiz', 19)
let aat = new Alumno('Ana', 'Abad Tudela', 23)
console.log(cpo < aat)      // imprime true ya que 19<23
```

## Métodos get y set en JS

Los métodos get y set no son necesarios, pero sí recomendables para modificar y/o acceder al contenido de las propiedades.

Se pueden definir para una o varias propiedades de una clase. Incluso puede definirse únicamente uno de estos métodos (set o get).

Get: lee la información (valor) de la propiedad.

Set: pone información en la propiedad.

Estos métodos no pueden llamarse igual que la propiedad, por eso pondremos propiedad (con guion bajo).

Con get y set no hay que poner paréntesis para llamar a los métodos:

Una vez definidos los métodos get y set para una propiedad, no usaremos paréntesis para invocar a esos métodos, accederemos como a sus propiedades:

- Para invocar al método get y que devuelva su valor haremos `= objeto.propiedad`

Para llamar al método set haremos: `objeto.propiedad = valor` Veamos un ejemplo para la propiedad nombre de la clase Alumno:

```
class Alumno {
  constructor(nombre, apellidos, edad) {
    this._nombre = nombre;
    this._apellidos = apellidos;
    this.edad = edad;
  }

  get nombre(){
    ...
  }
}
```

```

return this._nombre;
}
set nombre(nombre){
this._nombre = nombre;
}
}

let cpo = new Alumno('Carlos', 'Pérez Ortiz', 19);
cpo.nombre = 'Carlos Luis'; //set nombre('Carlos Luis')
console.log (cpo.nombre); // get nombre() -> Carlos Luis

```

El método **set** es más necesario en el caso que queramos que se guarde la información con un formato determinado. Dentro del método **set** podemos hacer los cambios que necesitemos.

Por ejemplo, para que se guarde siempre en mayúsculas:

```

set nombre(nombre){
this._nombre = nombre.toUpperCase(); //
}

```

El método **get** es muy útil para devolver información en un formato determinado como si accediéramos a una propiedad (sin paréntesis).

Por ejemplo, para devolver en formato: Apellidos, nombre utilizaremos el método `get nombreCompleto()`

```

get nombreCompleto(){
return this._apellidos + ', ' + this.nombre;
}

console.log(cpo.nombreCompleto); // se accede sin paréntesis

```

Get y set, en conjunto, pueden ser una forma muy eficaz de obtener seguridad en cuando a validación de datos.

### Agregar propiedades y métodos a los objetos de una clase (**prototype**).

En Javascript un objeto se crea a partir de otro (al que se llama *prototipo*). Así se crea una cadena de prototipos, el primero de los cuales es el objeto *null*.

Ya vimos como añadir una propiedad a un objeto. Pero si queremos que esa propiedad la tengan todos los objetos de la clase, y no queremos modificar el constructor, hay que hacerlo con **prototype**. Se le puede aplicar un valor por defecto.

Igualmente se pueden añadir métodos a la clase (también con **prototype**).

En el siguiente ejemplo se ha añadido la propiedad `email` y el método `ApellidosMayusc()` a la clase `Alumnos`.

```

class Alumno {
constructor(nombre, apellidos, edad) {
  this.nombre = nombre;
  this.apellidos = apellidos;
  this.edad = edad;
}
}

let cpo = new Alumno('Carlos', 'Pérez Ortiz', 19);

```

```
Alumno.prototype.email='...@email.com'; // valor por defecto
console.log(cpo.email) // ...@email.com

Alumno.prototype.ApellidosMayusc= function () {
    return this.apellidos.toLowerCase();
}

console.log(cpo.ApellidosMayusc());
```

## Prototipos y POO en JS5

Las versiones de Javascript anteriores a ES2015 no soportan clases ni herencia.

Este apartado está sólo para que comprendamos este código si lo vemos en algún programa, pero nosotros programaremos como hemos visto antes.

Si queremos emular en JS5 el comportamiento de las clases, para crear el constructor se crea una función con el nombre del objeto y para crear los métodos se aconseja hacerlo en el *prototipo* del objeto para que no se cree una copia del mismo por cada instancia que creemos:

```
function Alumno(nombre, apellidos, edad) {
    this.nombre = nombre
    this.apellidos = apellidos
    this.edad = edad
}
Alumno.prototype.getInfo = function() {
    return 'El alumno ' + this.nombre + ' ' + this.apellidos + ' tiene ' + this.edad + ' años'
}

let cpo = new Alumno('Carlos', 'Pérez Ortiz', 19)
console.log(cpo.getInfo()) // imprime 'El alumno Carlos Pérez Ortiz tiene 19 años'
```

Cada objeto tiene un prototipo del que hereda sus propiedades y métodos (es el equivalente a su clase, pero en realidad es un objeto que está instanciado).

Si añadimos una propiedad o método al prototipo se añade a todos los objetos creados a partir de él, lo que ahorra mucha memoria.

## Uso de call y apply en Javascript

Con la palabra reservada **call** se puede llamar a un método de un objeto, desde otro objeto que no tiene ese método.

```
let cpo= {
    nombre: 'Carlos',
    apellidos: 'Pérez Ortiz',
    nombreCompleto: function(){
        return this.apellidos + ', ' + this.nombre;
    }
}
```

```
let azp= {
    nombre: 'Ana',
    apellidos: 'Zubiri Peláez',
}

console.log(cpo.nombreCompleto()); // Pérez Ortiz, Carlos
console.log(cpo.nombreCompleto.call(azp)); // Zubiri Peláez, Ana
```

Se pueden pasar argumentos a la llamada con **call**, en ese caso **la función llamada** tratará los argumentos como un parámetro normal.

```
console.log(cpo.nombreCompleto.call(azp, argum1, argum2));
```

Con **apply** se puede llamar a un método de un objeto con los datos de otro objeto que no tiene definido ese método. Se diferencia de call en cómo se pasan los argumentos.

En el caso de apply los argumentos se pasan en un array, no de uno en uno.

```
let cpo= {
    nombre: 'Carlos',
    apellidos: 'Pérez Ortiz',
    nombreCompleto: function(titulo, email){
        return  titulo + ' ' + this.nombre + ', ' + email;
    }
}

let azp= {
    nombre: 'Ana',
    apellidos: 'Zubiri Peláez',
}

arrayArgumentos= ['Doña', 'anzupe@email.com'];
console.log(cpo.nombreCompleto.apply(azp, arrayArgumentos));
```

## La clase Object

Como todas las clases heredan de la clase Object, podemos utilizar con todos los objetos las propiedades y métodos de Object:

Ejemplo que lista los valores las propiedades de un objeto, también los métodos:

```
let cpo= {
    nombre: 'Carlos',
    apellidos: 'Pérez Ortiz',
    nombreCompleto: function(titulo, email){
        return  titulo + ' ' + this.nombre + ', ' + email;
    }
}

console.log(Object.values(cpo)); // (3) ['Carlos', 'Pérez Ortiz', f]
```

Ejemplo que lista propiedades (clave: valor) de las propiedades y métodos de un objeto. En este caso, se utiliza `Object.prototype`, que permite añadir métodos a todas las clases:

```
Object.prototype.imprime = function() {
    Object.entries(this).forEach(([key, value])=>{
        console.log(key + ': ' + value);
    })
}

let cpo= {
    nombre: 'Carlos',
    apellidos: 'Pérez Ortiz',
    nombreCompleto: function(titulo, email){
        return titulo + ' ' + this.nombre + ', ' + email;
    }
}

cpo.imprime();
```

Podemos utilizar el método `hasOwnProperty("nombre_propiedad")` para que

```
console.log(cpo.hasOwnProperty("edad")); // devuelve false porque "edad" no es una propiedad
```

Este método está heredado de **Object**. Podemos ver otros métodos del prototipo Object expandiendo la salida del objeto desde la consola:

```
console.log(cpo);
```

```
false
▼ {nombre: 'Carlos', apellidos: 'Pérez Ortiz', nombreCompleto: f} ⓘ
  apellidos: "Pérez Ortiz"
  nombre: "Carlos"
▶ nombreCompleto: f (titulo, email)
▼ [[Prototype]]: Object
  ▶ imprime: f ()
  ▶ constructor: f Object()
  ▶ hasOwnProperty: f hasOwnProperty()
  ▶ isPrototypeOf: f isPrototypeOf()
  ▶ propertyIsEnumerable: f propertyIsEnumerable()
  ▶ toLocaleString: f toLocaleString()
  ▶ toString: f toString()
  ▶ valueOf: f valueOf()
  ▶ __defineGetter__: f __defineGetter__()
  ▶ __defineSetter__: f __defineSetter__()
  ▶ __lookupGetter__: f __lookupGetter__()
  ▶ __lookupSetter__: f __lookupSetter__()
  ▶ __proto__: ...
  ▶ get __proto__: f __proto__()
  ▶ set __proto__: f __proto__()
```



## DWEC - Javascript Web Cliente.

# Anexo JavaScript - Clases Modulares

Anexo JavaScript - Clases Modulares.....	1
En index.html .....	1
El fichero clasePersona.js.....	1
El fichero clase heredada claseEmpleado.js.....	1
En app.js.....	2
Consideraciones sobre los módulos: .....	2

## En index.html

Hay que indicar que el archivo.js sea tipo “module”

```
<body>
    <h1>Clases heredadas</h1>
    <script type="module" src="js/app.js"></script>
</body>
</html>
```

Ojo, a partir de ahora, Javascript está en modo estricto (similar a ‘use strict’). Además, en el navegador existe el alcance (scope) independiente de nivel superior para cada `<script type="module">` en el documento HTML.

## El fichero clasePersona.js

Debe exportar la clase:

```
export class Persona{
    constructor(nombre, apellido, edad){
        this.nombre=nombre;
        this.apellido=apellido;
        this.edad=edad;
    }
}
```

## El fichero clase heredada claseEmpleado.js

Debe importar la clase padre y exportar su clase:

```
import {Persona} from './clasePersona.js'
export class Empleado extends Persona{
    constructor(nombre, apellido, edad, sueldo){
```

```

        super(nombre, apellido, edad);
        this.sueldo=sueldo;
    }
}

```

## En app.js

Importamos todas las clases que vayamos a utilizar:

```

import { Persona } from "./clases/clasePersona.js";
import { Empleado } from "./clases/claseEmpleado.js";

```

# Consideraciones sobre los módulos:

Para resumir, los conceptos centrales son:

1. Un módulo es un archivo. Para que funcione import/export, los navegadores necesitan <script type="module">.
2. Se pueden exportar (e importar) variables, funciones...
3. Los módulos tienen varias diferencias con los scripts convencionales:
  - o Diferido por defecto.
  - o Async funciona en scripts en línea.
  - o Para cargar scripts externos de otro origen (dominio/protocolo/puerto), se necesitan encabezados CORS.
  - o Se ignoran los scripts externos duplicados.
4. Los módulos tienen su propio alcance local de alto nivel y funcionalidad de intercambio a través de 'import/export'.
5. Los módulos siempre funcionan en modo estricto.
6. El código del módulo se ejecuta solo una vez. Las exportaciones se crean una vez y se comparten entre los importadores.

Como efecto secundario, los módulos siempre “ven” la página HTML completamente cargada, incluidos los elementos HTML debajo de ellos.

**Los módulos funcionan solo a través de HTTP(s), no localmente.** Si se intenta abrir una página web localmente a través del protocolo file://, se encontrará que las directivas import y export no funcionan. Hay que utilizar un servidor web local, o como VS Code [Live Server Extension](#) para probar los módulos.



## DWEC – Javascript Web Cliente.

# Anexo JavaScript – Exportar Funciones Librería vs Prototipo

Anexo JavaScript – Exportar Funciones Librería vs Prototipo .....	1
En index.html .....	1
El fichero libreria.js .....	1
El fichero ejemploUsoLibreria.js .....	2
Resultado: .....	2

## En index.html

Hay que indicar que el archivo.js sea tipo “module”

```
<body>
  <h1>Clases heredadas</h1>
  <script type="module" src="ejemploUsoLibreria.js"></script>
</body>
</html>
```

Ojo, a partir de ahora, hay algunas restricciones en cuanto al lenguaje Javascript

## El fichero libreria.js

Se deben exportar las funciones:

```
// en lugar de usar funciones prototipo se pueden usar funciones exportadas
Number.prototype.moneda=function(){
    return this.toLocaleString( undefined, { style: 'currency', currency: 'EUR' },
    {minimumFractionDigits: 2, maximumFractionDigits: 2} );
}

String.prototype.nombrePropio=function(){
    return this[0].toUpperCase() + this.toLowerCase().slice(1);
}

//ejemplo de uso: console.log("abCDE".nombrePropio());

// FUNCIONES PARA EXPORTAR
function formatearANombrePropio(string) {
    return string[0].toUpperCase() + string.slice(1).toLowerCase();
}
```

```

}

function formatearAFechaLocal(fecha) {
  return fecha.toLocaleDateString();
}

function formatearASueldo(sueldo) {
  return sueldo.toLocaleString("es-ES", {
    style: "currency",
    currency: "EUR",
  });
}

export const funcionesLibreria = {
  formatearAFechaLocal,
  formatearANombrePropio,
  formatearASueldo,
};

// export { funciones } from "./funciones.js";

```

## El fichero ejemploUsoLibreria.js

Debe importar el objeto con todas las funciones que queramos tener disponibles:

```

import {funcionesLibreria} from "./libreria.js";

console.log(funcionesLibreria.formatearANombrePropio("feDERICO"));
console.log(funcionesLibreria.formatearAFechaLocal(new Date()));
console.log(funcionesLibreria.formatearASueldo(123456.789));

```

## Resultado:

Federico	<a href="#">ejemploUsoLibreria.js:3</a>
2/11/2023	<a href="#">ejemploUsoLibreria.js:4</a>
123.456,79 €	<a href="#">ejemploUsoLibreria.js:5</a>



## DWEC – Javascript Web Cliente.

# Anexo JavaScript – Variables Globales exportadas

Fichero con las variables globales:	.....	1
Importar el objeto en los .js que queramos	.....	1

### Fichero con las variables globales:

Creamos uno o más objetos con las variables globales que queramos.

```
js > libreria > JS variables_globales.js > ...
1  let objetoVariables={
2    variable1:0,
3    variable2:"lo que sea",
4    variable3: true,
5  }
6
7  export {objetoVariables};
```

### Importar el objeto en los .js que queramos

En los archivos .js que vayamos a utilizar las variables globales, importamos el objeto:

```
import { objetoVariables } from "./variables_globales.js";
```

Para referenciar a una variable, usaremos la notación: objeto.propiedad

```
if(objetoVariables.variable1 > 5){

}
```



## DWEC – Javascript Web Cliente.

JavaScript 04 – DOM - Document Object Model .....	1
Introducción.....	1
Acceso a los nodos.....	3
getElementById(id) .....	3
getElementsByClassName(clase).....	4
getElementsByTagName(elemento) .....	4
querySelector(selector).....	4
querySelectorAll(selector).....	4
Atajos .....	5
Acceso a nodos a partir de otros .....	6
Propiedades de un nodo.....	7
innerHTML .....	7
textContent .....	7
value.....	7
Manipular el árbol DOM .....	8
createElement.....	8
createTextNode .....	8
appendChild .....	8
insertBefore .....	8
removeChild.....	8
replaceChild .....	9
cloneNode .....	9
Ejemplo de creación de nuevos nodos: .....	9
Modificar el DOM con ChildNode .....	10
Atributos de los nodos .....	10
Estilos de los nodos .....	11
Atributos de clase .....	11

# JavaScript 04 – DOM - Document Object Model

## Introducción

La mayoría de las veces que se programa con Javascript es para que se ejecute en una página web mostrada por el navegador. En este contexto, y para facilitar el desarrollo de la aplicación, tenemos acceso a ciertos objetos que nos permiten interactuar con la página (DOM) y con el navegador (Browser Object Model, BOM).

El **DOM** es una estructura en árbol que representa todos los elementos HTML de la página y sus atributos.

Todo lo que contiene la página se representa como nodos del árbol y mediante el DOM podemos acceder a cada nodo, modificarlo, eliminarlo o añadir nuevos nodos de forma que cambiamos dinámicamente la página mostrada al usuario.

La raíz del árbol DOM es **document** y de este nodo cuelgan el resto de elementos HTML.

Cada elemento constituye su propio nodo y tiene subnodos con sus *atributos*, *estilos* y otros elementos HTML que contiene.

Nota: Un **elemento HTML** consiste en:

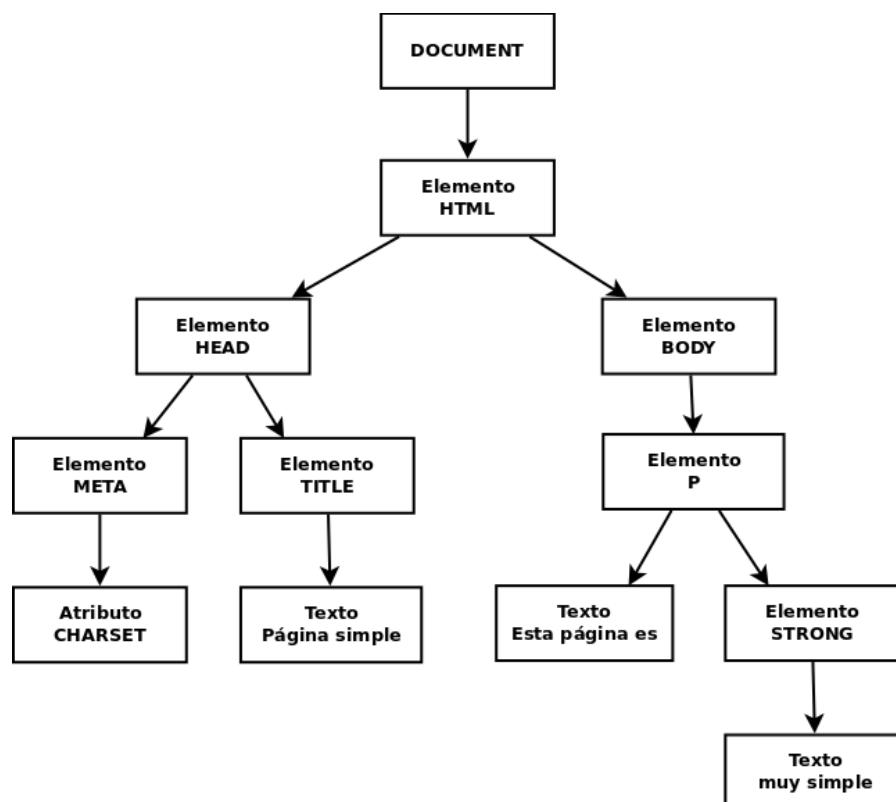
- Una **etiqueta inicial**. Opcionalmente contiene pares “atributo: valor”.
- **Contenido** del elemento (no siempre aparece).
- **Etiqueta final** o de cierre (no siempre aparece).

Es frecuente que alguien se refiera a un ‘elemento HTML’ como ‘etiqueta HTML’. Por lo que debes interpretar el significado de **etiqueta** en cada momento según el contexto.

Por ejemplo, la página HTML:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>Página simple</title>
</head>
<body>
  <p>Esta página es <strong>muy simple</strong></p>
</body>
</html>
```

se convierte en el siguiente árbol DOM:



Cada elemento HTML suele originar 2 nodos:

- **Element**: correspondiente al elemento.
- **Text**: correspondiente a su contenido (lo que hay entre la etiqueta inicial y su par de cierre)

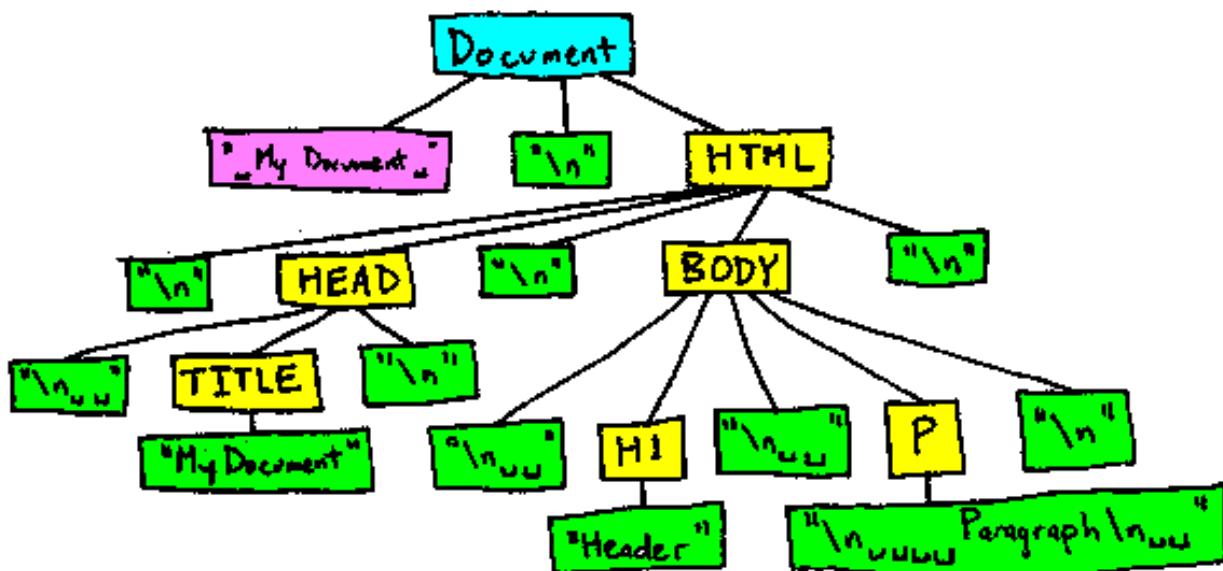
Cada nodo es un objeto con sus propiedades y métodos.

El ejemplo anterior está simplificado porque sólo aparecen los nodos de tipo **elemento**, pero en realidad también generan nodos los saltos de línea, tabuladores, espacios, comentarios, etc.

En el siguiente ejemplo podemos ver TODOS los nodos que realmente se generan. La página:

```
<!DOCTYPE html>
<html>
<head>
  <title>My Document</title>
</head>
<body>
  <h1>Header</h1>
  <p>
    Paragraph
  </p>
</body>
</html>
```

se convierte en el siguiente árbol DOM:



## Acceso a los nodos

Los principales métodos para acceder a los diferentes nodos son:

**getElementById(id)**

.**getElementById(id)**: devuelve el nodo con la *id* pasada. Ej.:

```
let nodo = document.getElementById('main'); // nodo contendrá el nodo cuya id es _main_
```

Cuidado de no confundir con el método `.getElementsByName(nombre)`, que devuelve los elementos con el atributo *name* especificado.

### getElementsByClassName(clase)

**.getElementsByClassName(clase):** devuelve una colección (similar a un array) con todos los nodos de la *clase* indicada. Ej.:

```
let nodos = document.getElementsByClassName('error'); // nodos contendrá todos los nodos cuya clase es _error_
```

NOTA: las colecciones son similares a arrays (se accede a sus elementos con *[indice]*) pero no se les pueden aplicar los métodos *filter, map, ...* a menos que se conviertan a arrays con *Array.from()*

### getElementsByTagName(elemento)

**.getElementsByTagName(elemento):** devuelve una colección con todos los nodos del tipo *elemento* HTML indicado. Ej.:

```
let nodos = document.getElementsByTagName('p'); // nodos contendrá todos los nodos de tipo _<p>_
```

### querySelector(selector)

**.querySelector(selector):** devuelve el primer nodo seleccionado por el *selector CSS* indicado. Ej.:

```
let nodo = document.querySelector('p.error'); // nodo contendrá el primer párrafo de clase _error_
```

### querySelectorAll(selector)

**.querySelectorAll(selector):** devuelve una NodeList con todos los nodos seleccionados por el *selector CSS* indicado. Ej.:

```
let nodos = document.querySelectorAll('p.error'); // nodos contendrá todos los párrafos de clase _error_
```

```
let nodo = document.querySelectorAll('#text'); // nodo contendrá el elemento con ID=text
```

Más información en: <https://developer.mozilla.org/es/docs/Web/API/Document/querySelectorAll>

**NOTA:** Los objetos `NodeList` son colecciones de nodos como los devueltos por propiedades como `Node.childNodes` y el método `document.querySelectorAll()` ..

**NOTA:** al aplicar estos métodos sobre `document` se seleccionarán sobre la página (objeto `document`). Pero podrían también aplicarse a cualquier nodo, en ese caso la búsqueda se realizaría sólo entre los descendientes de dicho nodo.

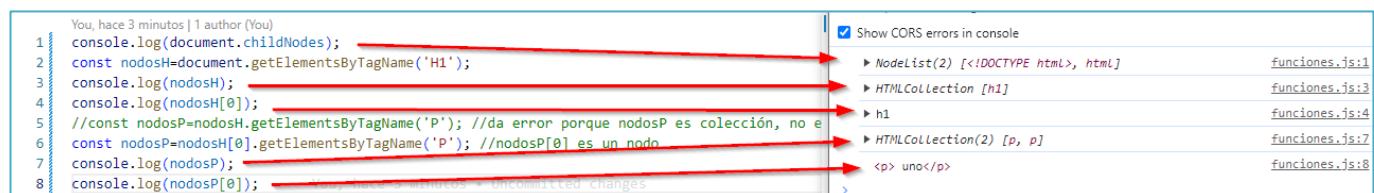
## Ejemplo:

Partiendo de este .html:

```
<body>
  <h1>El DOM
    <p> uno</p>
    <p> dos</p>
  </h1>
  <script src="js/funciones.js"></script>
</body>
```

Aplicamos el siguiente código:

```
console.log(document.childNodes);
const nodosH=document.getElementsByTagName('H1');
console.log(nodosH);
console.log(nodosH[0]);
//const nodosP=nodosH.getElementsByTagName('P'); //da error porque nodosP es colección, no es nodo
const nodosP=nodosH[0].getElementsByTagName('P'); //nodosP[0] es un nodo
console.log(nodosP);
console.log(nodosP[0]);
```



## Atajos

También tenemos ‘atajos’ para obtener algunos elementos comunes:

- document.documentElement: devuelve el nodo del elemento `<html>`
- document.head: devuelve el nodo del elemento `<head>`
- document.body: devuelve el nodo del elemento `<body>`
- document.title: devuelve el nodo del elemento `<title>`
- document.link: devuelve una colección con todos los hiperenlaces del documento
- document.anchor: devuelve una colección con todas las anclas del documento
- document.forms: devuelve una colección con todos los formularios del documento
- document.images: devuelve una colección con todas las imágenes del documento
- document.scripts: devuelve una colección con todos los scripts del documento

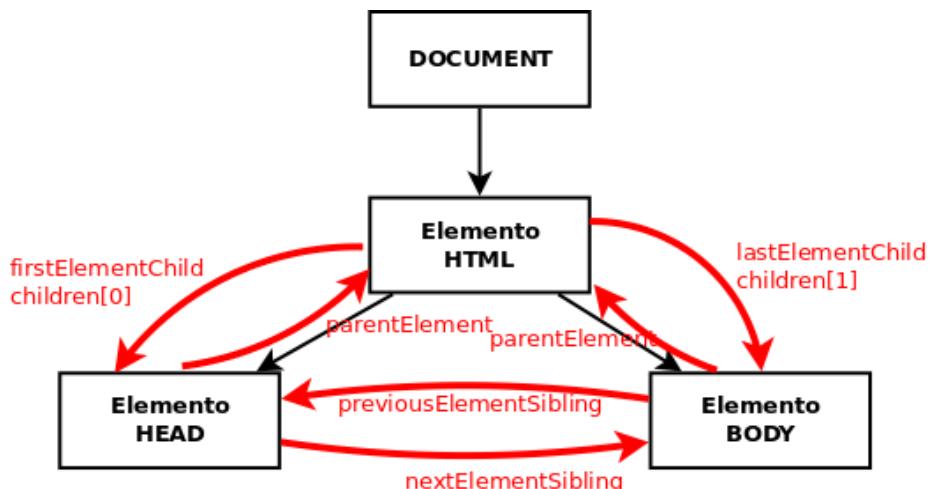
**EJERCICIO:** Para hacer los ejercicios de este tema descárgate [esta página de ejemplo](#) y ábrelo en tu navegador. Obtén por consola, al menos de 2 formas diferentes lo que se pide:

1. El elemento con id ‘input2’
2. La colección de párrafos
3. Lo mismo pero sólo de los párrafos que hay dentro del div ‘lipsum’
4. El formulario (ojo, no la colección con el formulario sino sólo el formulario)

5. Todos los inputs
6. Sólo los inputs con nombre 'sexo'
7. Los items de lista de la clase 'important' (sólo los LI)

## Acceso a nodos a partir de otros

En muchas ocasiones queremos acceder a cierto nodo a partir de uno dado. Para ello tenemos los siguientes métodos que se aplican sobre un elemento del árbol DOM:



- `elemento.parentElement`: devuelve el elemento padre de *elemento*
- `elemento.children`: devuelve la colección con todos los elementos hijo de *elemento* (sólo elementos HTML, no comentarios ni nodos de tipo texto)
- `elemento.childNodes`: devuelve la colección con todos los hijos de *elemento*, incluyendo comentarios y nodos de tipo texto por lo que no suele utilizarse
- `elemento.firstElementChild`: devuelve el elemento HTML que es el primer hijo de *elemento*
- `elemento.firstChild`: devuelve el nodo que es el primer hijo de *elemento* (incluyendo nodos de tipo texto o comentarios)
- `elemento.lastElementChild`, `elemento.lastChild`: igual pero con el último hijo
- `elemento.nextElementSibling`: devuelve el elemento HTML que es el siguiente hermano de *elemento*
- `elemento.nextSibling`: devuelve el nodo que es el siguiente hermano de *elemento* (incluyendo nodos de tipo texto o comentarios)
- `elemento.previousElementSibling`, `elemento.previousSibling`: igual pero con el hermano anterior
- `elemento.hasChildNodes`: indica si *elemento* tiene o no nodos hijos
- `elemento.childElementCount`: devuelve el nº de nodos hijo de *elemento*

**IMPORTANTE:** a menos que interesen comentarios, saltos de página, etc ..., **siempre** hay que usar los métodos que sólo devuelven elementos HTML, no todos los nodos.

EJERCICIO: Siguiendo con la [página de ejemplo](#) obtén desde la consola, al menos de 2 formas diferentes:

1. El primer párrafo que hay dentro del div 'lipsum'
2. El segundo párrafo de 'lipsum'
3. El último item de la lista
4. El elemento *label* de 'Escoge sexo'

## Propiedades de un nodo

Las principales propiedades de un nodo son:

### innerHTML

`elemento.innerHTML`: todo lo que hay entre la etiqueta que abre *elemento* y la que lo cierra, incluyendo otras etiquetas HTML. Por ejemplo, si *elemento* es el nodo:

```
<p>Esta página es <strong>muy simple</strong></p>
```

```
let contenido = elemento.innerHTML; // contenido='Esta página es <strong>muy simple</strong>'
```

### textContent

`elemento.textContent`: todo lo que hay entre la etiqueta que abre *elemento* y la que lo cierra, pero ignorando otras etiquetas HTML. Siguiendo con el ejemplo anterior:

```
let contenido = elemento.textContent; // contenido='Esta página es muy simple'
```

### value

`elemento.value`: devuelve la propiedad ‘value’ de un `<input>` (en el caso de un `<input>` de tipo `text` devuelve lo que hay escrito en él).

Como los `<inputs>` no tienen etiqueta de cierre (`</input>`) no podemos usar `.innerHTML` ni `.textContent`.

Por ejemplo:

si *elem1* es el nodo `<input name="nombre">` y *elem2* es el nodo `<input type="radio" value="H"> Hombre`

```
let cont1 = elem1.value; // cont1 valdría lo que haya escrito en el <input> en ese momento
let cont2 = elem2.value; // cont2="H"
```

### Otras propiedades:

- `elemento.innerText`: igual que `textContent`
- `elemento.focus`: pone (sitúa) el foco en *elemento* (para inputs, etc). Para quitarle el foco `elemento.blur`
- `elemento.clientHeight / elemento.clientWidth`: devuelve el alto / ancho visible del *elemento*
- `elemento.offsetHeight / elemento.offsetWidth`: devuelve el alto / ancho total del *elemento*
- `elemento.clientLeft / elemento.clientTop`: devuelve la distancia de *elemento* al borde izquierdo / borde superior
- `elemento.offsetLeft / elemento.offsetTop`: devuelve los *píxeles* que hemos desplazado *elemento* a la izquierda / abajo

EJERCICIO: Obtén desde la consola, al menos de 2 formas:

1. El `innerHTML` de la etiqueta de ‘Escoge sexo’
2. El `textContent` de esa etiqueta
3. El valor del primer input de sexo

4. El valor del sexo que esté seleccionado (difícil, búscalo por Internet)

## Manipular el árbol DOM

Vamos a ver qué métodos nos permiten cambiar el árbol DOM, y por tanto modificar la página:

### createElement

`document.createElement('etiqueta')`: crea un nuevo elemento HTML con la etiqueta indicada, pero aún no se añade a la página. Ej.:

```
let nuevoLi = document.createElement('li');
```

### createTextNode

`document.createTextNode('texto')`: crea un nuevo nodo de texto con el texto indicado, que luego tendremos que añadir a un nodo HTML. Ej.:

```
let textoLi = document.createTextNode('Nuevo elemento de lista');
```

### appendChild

`elemento.appendChild(nuevoNodo)`: añade *nuevoNodo* como último hijo de *elemento*. Ahora ya se ha añadido a la página. Ej.:

```
nuevoLi.appendChild(textoLi);      // añade el texto creado al elemento LI creado
let miPrimeraLista = document.getElementsByTagName('ul')[0]; // selecciona el 1º UL de
la página
miPrimeraLista.appendChild(nuevoLi); // añade LI como último hijo de UL, es decir al
final de la Lista
```

### insertBefore

`elemento.insertBefore(nuevoNodo, nodo)`: añade *nuevoNodo* como hijo de *elemento* antes del hijo *nodo*. Ej.:

```
let miPrimeraLista = document.getElementsByTagName('ul')[0]; // selecciona el 1º UL de
la página
let primerElementoDeLista = miPrimeraLista.getElementsByTagName('li')[0]; // selecciona
el 1º LI de miPrimeraLista
miPrimeraLista.insertBefore(nuevoLi, primerElementoDeLista); // añade LI al principio
de la Lista
```

### removeChild

`elemento.removeChild(nodo)`: borra *nodo* de *elemento* y por tanto se elimina de la página. Ej.:

```
let miPrimeraLista = document.getElementsByTagName('ul')[0]; // selecciona el 1º UL de
la página
let primerElementoDeLista = miPrimeraLista.getElementsByTagName('li')[0]; // selecciona
el 1º LI de miPrimeraLista
```

```
miPrimeraLista.removeChild(primerElementoDeLista); // borra el primer elemento de La
lista
// También podríamos haberlo borrado sin tener el parent con:
primerElementoDeLista.parentElement.removeChild(primerElementoDeLista);
```

## replaceChild

elemento.replaceChild(nuevoNodo, viejoNodo): reemplaza *viejoNodo* con *nuevoNodo* como hijo de *elemento*. Ej.:

```
let miPrimeraLista = document.getElementsByTagName('ul')[0]; // selecciona el 1º UL de
la página
let primerElementoDeLista = miPrimeraLista.getElementsByTagName('li')[0]; // selecciona
el 1º LI de miPrimeraLista
miPrimeraLista.replaceChild(nuevoLi, primerElementoDeLista); // reemplaza el 1º
elemento de la lista con nuevoLi
```

## cloneNode

elementoAClonar.cloneNode(boolean): devuelve un clon de *elementoAClonar* o de *elementoAClonar* con todos sus descendientes según le pasemos como parámetro false o true. Luego podremos insertarlo donde queramos.

**OJO:** Si añado con el método appendChild un nodo que estaba en otro sitio **se elimina de donde estaba** para añadirse a su nueva posición. Si quiero que esté en los 2 sitios deberá clonar el nodo y luego añadir el clon y no el nodo original.

## Ejemplo de creación de nuevos nodos:

Tenemos un código HTML con un DIV que contiene 3 párrafos y vamos a añadir un nuevo párrafo al final del div con el texto ‘Párrafo añadido al final’ y otro que sea el 2º del div con el texto ‘Este es el **nuevo** segundo párrafo’:

```
<div id="articulos">
    <p>Este es el primer párrafo que tiene <strong>algo en negrita</strong>.</p>
    <p>Este era el segundo párrafo pero será desplazado hacia abajo.</p>
    <p>Y este es el último párrafo pero luego añadiremos otro después</p>
</div>
```

```
let miDiv=document.getElementById('articulos');

miDiv.innerHTML+="

Párrafo añadido al final

";

let nuevoSegundoParrafo=document.createElement('p');
nuevoSegundoParrafo.innerHTML='Este es el <strong>nuevo</strong> segundo párrafo';

let segundoParrafo=miDiv.children[1];
miDiv.insertBefore(nuevoSegundoParrafo, segundoParrafo);
```

Resultado:

```
Este es el primer párrafo que tiene algo en negrita.  
Este es el nuevo segundo párrafo  
Este era el segundo párrafo pero será desplazado hacia abajo.  
Y este es el último párrafo pero luego añadiremos otro después  
Párrafo añadido al final
```

Si utilizamos la propiedad **innerHTML** el código a usar es mucho más simple:

```
let ultimoParrafo = document.createElement('p');  
ultimoParrafo.innerHTML = 'Párrafo añadido al final';  
miDiv.appendChild(ultimoParrafo);
```

**OJO:** La forma de añadir el último párrafo (línea #3: `miDiv.innerHTML+= '<p>Párrafo añadido al final</p>';`) aunque es válida, no es muy eficiente ya que obliga al navegador a volver a pintar TODO el contenido de miDIV. La forma correcta de hacerlo sería:

```
let ultimoParrafo = document.createElement('p');  
ultimoParrafo.innerHTML = 'Párrafo añadido al final';  
miDiv.appendChild(ultimoParrafo);
```

Así sólo debe repintar el párrafo añadido, conservando todo lo demás que tenga `miDiv`.

Podemos ver más ejemplos de creación y eliminación de nodos en [W3Schools](#).

EJERCICIO: Añade a la página:

1. Un nuevo párrafo al final del DIV '`lipsum`' con el texto “Nuevo párrafo **añadido** por javascript” (fíjate que una palabra está en negrita)
2. Un nuevo elemento al formulario tras el '`Dato 1`' con la etiqueta '`Dato 1 bis`' y el INPUT con id '`input1bis`' que al cargar la página tendrá escrito “Hola”

## Modificar el DOM con ChildNode

Childnode es una interfaz que permite manipular del DOM de forma más sencilla pero no está soportada en los navegadores Safari de IOS. Incluye los métodos:

- `elemento.before(nuevoNodo)`: añade el `nuevoNodo` pasado, antes del nodo `elemento`
- `elemento.after(nuevoNodo)`: añade el `nuevoNodo` pasado, después del nodo `elemento`
- `elemento.replaceWith(nuevoNodo)`: reemplaza el nodo `elemento` con el `nuevoNodo` pasado
- `elemento.remove()`: elimina el nodo `elemento`

## Atributos de los nodos

Podemos ver y modificar los valores de los atributos de cada elemento HTML y también añadir o eliminar atributos:

- `elemento.attributes`: devuelve un array con todos los atributos de `elemento`

- `elemento.hasAttribute('nombreAtributo')`: indica si *elemento* tiene o no definido el atributo *nombreAtributo*
- `elemento.getAttribute('nombreAtributo')`: devuelve el valor del atributo *nombreAtributo* de *elemento*. Para muchos elementos este valor puede directamente con `elemento.getAttribute`.
- `elemento.setAttribute('nombreAtributo', 'valor')`: establece *valor* como nuevo valor del atributo *nombreAtributo* de *elemento*. También puede cambiarse el valor directamente con `elemento.setAttribute=valor`.
- `elemento.removeAttribute('nombreAtributo')`: elimina el atributo *nombreAtributo* de *elemento*

A algunos atributos comunes como `id`, `title` o `className` (para el atributo `class`) se puede acceder y cambiar como si fueran una propiedad del elemento (`elemento.getAttribute`). Ejemplos:

```
let miPrimeraLista = document.getElementsByTagName('ul')[0]; // selecciona el 1º UL de la página
miPrimeraLista.id = 'primera-lista';
// es equivalente ha hacer:
miPrimeraLista.setAttribute('id', 'primera-lista');
```

## Estilos de los nodos

Los estilos están accesibles como el atributo `style`. Cualquier estilo es una propiedad de dicho atributo, pero con la sintaxis *camelCase* en vez de *kebab-case*.

Por ejemplo, para cambiar el color de fondo (propiedad `background-color`) y poner el color rojo al elemento `miPrimeraLista` haremos:

```
miPrimeraLista.style.backgroundColor = 'red';
```

De todas formas, normalmente **NO CAMBIAREMOS ESTILOS** a los elementos, si no que les pondremos o quitaremos clases que ya estén en el CSS, que harán que se le apliquen o no los estilos definidos en esas clases.

## Atributos de clase

Ya sabemos que el aspecto de la página debe configurarse en el CSS, por lo que no debemos aplicar atributos `style` al HTML. En lugar de ello pondremos clases a los elementos que harán que se les aplique el estilo definido para dicha clase.

Como es algo muy común, en lugar de utilizar las instrucciones de `elemento.setAttribute('className', 'destacado')` o directamente `elemento.className='destacado'` podemos usar la propiedad `classList` que devuelve la colección de todas las clases que tiene el elemento.

Por ejemplo, si *elemento* es `<p class="destacado direccion">....</p>`:

```
let clases=elemento.classList; // clases=['destacado', 'direccion'], OJO es una colección, no un Array
```

Además, dispone de los métodos:

*add*

**.add(clase)**: añade al elemento la clase pasada (si ya la tiene no hace nada). Ej.:

```
elemento.classList.add('primero'); // ahora elemento será <p class="destacado direccion primero">...
```

*remove*

**.remove(clase)**: elimina del elemento la clase pasada (si no la tiene no hace nada). Ej.:

```
elemento.classList.remove('direccion'); // ahora elemento será <p class="destacado primero">...
```

*toogle*

**.toogle(clase)**: añade la clase pasada si no la tiene o la elimina si la tiene ya. Ej.:

```
elemento.classList.toggle('destacado'); // ahora elemento será <p class="primero">...
elemento.classList.toggle('direccion'); // ahora elemento será <p class="primero direccion">...
```

*contains*

**.contains(clase)**: dice si el elemento tiene o no la clase pasada. Ej.:

```
elemento.classList.contains('direccion'); // devuelve true
```

*replace*

**.replace(oldClase, newClase)**: reemplaza del elemento una clase existente por una nueva. Ej.:

```
elemento.classList.replace('primero', 'ultimo'); // ahora elemento será <p class="ultimo direccion">...
```

Ten en cuenta que NO todos los navegadores soportan *classList* por lo que si queremos añadir o quitar clases en navegadores que no lo soportan debemos hacerlo con los métodos estándar. Por ejemplo, para añadir la clase 'rojo':

```
let clases = elemento.className.split(" ");
if (clases.indexOf('rojo') == -1) {
  elemento.className += ' ' + 'rojo';
}
```

## DWEC – Javascript Web Cliente.

JavaScript 05 – Browser Object Model (BOM).....	1
Introducción.....	1
Timers .....	1
Objetos del BOM.....	3
Objeto window.....	3
Diálogos.....	4
Objeto location .....	4
Objeto history .....	5
Otros objetos .....	5

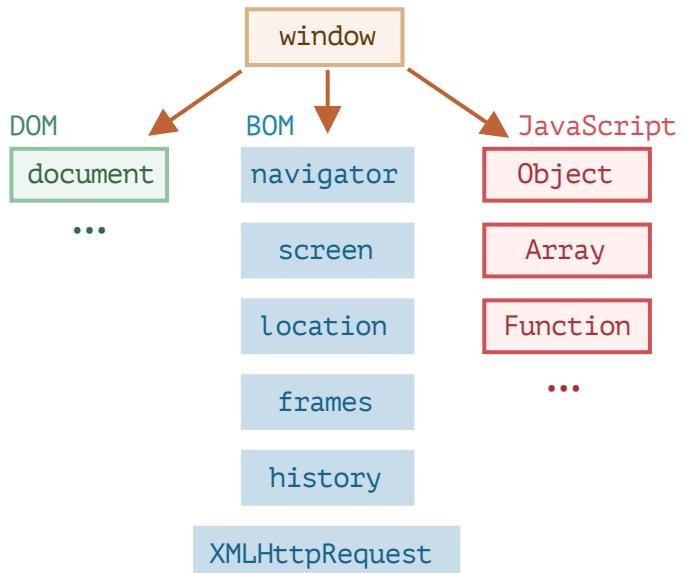
# JavaScript 05 – Browser Object Model (BOM)

## Introducción

Hemos visto cómo interactuar con la página (DOM), ahora veremos cómo acceder a objetos que nos permitan interactuar con el navegador utilizando el BOM (*Browser Object Model*).

Usando los objetos BOM podemos:

- Abrir, cambiar y cerrar ventanas.
- Ejecutar código transcurrido cierto tiempo (*timers*).
- Obtener información del navegador.
- Ver y modificar propiedades de la pantalla.
- Gestionar cookies, ...



## Timers

Permiten ejecutar código en el futuro (cuando transcurran los milisegundos indicados). Hay 2 tipos:

- `setTimeout(función, milisegundos)`: ejecuta la función especificada **una sola vez**, cuando transcurran los milisegundos indicados.
- `setInterval(función, milisegundos)`: ejecuta la función especificada **cada vez que transcurran** los milisegundos indicados, hasta que sea cancelado el *timer*.

Se pueden pasar más parámetros a las funciones, se ponen tras los milisegundos, y serán los parámetros que recibirá la función a ejecutar.

Ambas funciones devuelven un identificador que nos permitirá cancelar la ejecución del código, con:

- clearTimeout(identificador)
- clearInterval(identificador)

Ejemplo:

```
const idTimeout = setTimeout(() => console.log('Timeout que se ejecuta al cabo de 1 seg.'), 1000);

let i = 1;
const idInterval = setInterval(() => {
    console.log('Interval cada 3 seg. Ejecución nº: ' + i++);
    if (i === 5) {
        clearInterval(idInterval);
        console.log('Fin de la ejecución del Interval');
    }
}, 3000);
```

EJERCICIO: Prueba a ejecutar cada una de esas funciones.

En lugar de definir la función a ejecutar podemos llamar a una función que ya exista:

```
function showMessage() {
    console.log('Timeout que se ejecuta al cabo de 1 seg.')
}

const idTimeout=setTimeout(showMessage, 1000);
```

Pero en ese caso hay que poner sólo el nombre de la función, sin (), ya que si los ponemos se ejecutaría la función en ese momento y no transcurrido el tiempo indicado.

Si necesitamos pasar algún parámetro a la función, los añadiremos como parámetros de setTimeout o setInterval después del intervalo.

Ejemplo:

```
function showMessage(msg) {
    alert(msg)
}

const idTimeout = setTimeout(showMessage, 1000, 'Timeout que se ejecuta al cabo de 1 seg.');
```

Otro ejemplo:

```
function myCallback(a, b) {
    // Tu código debe ir aquí
    // Los parámetros son totalmente opcionales
    console.log(a);
    console.log(b);
}

const intervalID = setInterval(myCallback, 500, 'parámetro 1', 'parámetro 2');
```

## Objetos del BOM

Al contrario que para el DOM, no existe un estándar de BOM pero sus objetos (window, location, history, etc) son bastante parecidos en los diferentes navegadores.

### Objeto window

Representa la ventana del navegador y es el objeto principal. De hecho puede omitirse al llamar a sus propiedades y métodos, por ejemplo, el método `setTimeout()` es en realidad `window.setTimeout()`.

Sus principales propiedades y métodos son:

- `.name`: nombre de la ventana actual
- `.statusbar`: valor de la barra de estado
- `.screenX/.screenY`: distancia de la ventana a la esquina izquierda/superior de la pantalla
- `.outerWidth/.outerHeight`: ancho/alto total de la ventana, incluyendo la toolbar y la scrollbar
- `.innerWidth/.innerHeight`: ancho/alto útil del documento, sin la toolbar y la scrollbar
- `.open(url, nombre, opciones)`: abre una nueva ventana. Devuelve el nuevo objeto ventana.

Las principales **opciones de `.open()`** (lista de ítems separados por comas sin espacios) son:

- `.toolbar`: si tendrá barra de herramientas
- `.location`: si tendrá barra de dirección
- `.directories`: si tendrá botones Adelante/Atrás
- `.status`: si tendrá barra de estado
- `.menubar`: si tendrá barra de menú
- `.scrollbar`: si tendrá barras de desplazamiento
- `.resizable`: si se puede cambiar su tamaño
- `.width=px/.height=px`: ancho/alto
- `.left=px/.top=px`: posición izq/sup de la ventana

Más información en [https://www.w3schools.com/jsref/met\\_win\\_open.asp](https://www.w3schools.com/jsref/met_win_open.asp)

- `.opener`: referencia a la ventana desde la que se abrió esta ventana (para ventanas abiertas con `open`)
- `.close()`: la cierra (pide confirmación, a menos que la hayamos abierto con `open`)
- `.moveTo(x,y)`: la mueve a las coord indicadas
- `.moveBy(x,y)`: la desplaza los px indicados
- `.resizeTo(x,y)`: especifica el ancho y alto indicados
- `.resizeBy(x,y)`: añade esos pixeles de ancho/alto
- `.pageXOffset / pageYOffset`: scroll actual de la ventana horizontal / vertical
- Otros métodos: `.back()`, `.forward()`, `.home()`, `.stop()`, `.focus()`, `.blur()`, `.find()`, `.print()`, ...

Más información en [https://www.w3schools.com/jsref/obj\\_window.asp](https://www.w3schools.com/jsref/obj_window.asp)

NOTA: por seguridad no se puede mover una ventana fuera de la pantalla, ni darle un tamaño menor de 100x100 px. Tampoco se puede mover una ventana no abierta con `.open()`, o si tiene varias pestañas.

EJEMPLO:

- a) Abrir una nueva ventana de dimensiones 500x200px en la posición (200,100)
- b) Escribir en la nueva ventana (con `document.write`) un título H1 que diga ‘Hola javascritos del Claudio’.
- c) Al hacer clic sobre un botón de la ventana inicial, que la ventana se desplace 40 px a la izquierda y 50 hacia abajo
- d) Al hacer clic sobre otro botón de la ventana incicial, que se cierre la nueva ventana.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
<body>
  <form action="">
    <input type="button" value="Deslazar" onclick="desplazar()">
    <input type="button" value="Cerrar" onclick="cerrar()">
  </form>
  <script src="js/bom1.js"></script>
</body>
</html>
```

```
function desplazar(){
  nuevaVentana.moveBy(40,50);
}

const cerrar = ()=> {
  nuevaVentana.close();
}

const nuevaVentana=window.open("", "_blank", "width= 500,height=200,left=200,top=100");
nuevaVentana.document.write("<h1>Hola javascritos del Claudio</h1>");
```

EJERCICIO: Haz que a los 2 segundos de abrir la página se abra un *popup* con un mensaje de bienvenida. Esta ventana tendrá en su interior un botón Cerrar que permitirá que el usuario la cierre haciendo clic en él. Tendrá el tamaño justo para visualizar el mensaje y no tendrá barras de scroll, ni de herramientas, ni de dirección... únicamente el mensaje.

## Diálogos

Hay 3 métodos del objeto *window* que ya conocemos y que nos permiten abrir ventanas de diálogo con el usuario:

- *window.alert(mensaje)*: muestra un diálogo con el mensaje indicado y un botón de ‘Aceptar’
- *window.confirm(mensaje)*: muestra un diálogo con el mensaje indicado y botones de ‘Aceptar’ y ‘Cancelar’. Devuelve *true* si se ha pulsado el botón de aceptar del diálogo y *false* si no.
- *window.prompt(mensaje [, valor predeterminado])*: muestra un diálogo con el mensaje indicado, un cuadro de texto (vacío o con el valor predeterminado indicado) y botones de ‘Aceptar’ y ‘Cancelar’. Si se pulsa ‘Aceptar’ devolverá un *string* con el valor que haya en el cuadro de texto y si se pulsa ‘Cancelar’ o se cierra devolverá *null*.

## Objeto location

Contiene información sobre la URL actual del navegador y podemos modificarla. Sus principales propiedades y métodos son:

- `.href`: devuelve la URL actual completa
- `.protocol`, `.host`, `.port`: devuelve el protocolo, host y puerto respectivamente de la URL actual
- `.pathname`: devuelve la ruta al recurso actual
- `.reload()`: recarga la página actual
- `.assign(url)`: carga la página pasada como parámetro
- `.replace(url)`: ídem pero sin guardar la actual en el historial

**EJERCICIO:**

- muestra la ruta completa de la página actual
- muestra el servidor de esta página
- carga la página de Google usando el objeto `location`

```
<script>
    console.log( window.location.href);
    console.log(location.host);
    console.log(location.assign('https://www.google.es/'));
</script>
```

**Objeto history**

Permite acceder al historial de páginas visitadas y navegar por él:

- `.length`: muestra el número de páginas almacenadas en el historial
- `.back()`: vuelve a la página anterior
- `.forward()`: va a la siguiente página
- `.go(num)`: se mueve *num* páginas hacia adelante en el historial (si *num* es positivo), o hacia atrás (si *num* es negativo).

**EJERCICIO:** Vuelve a la página anterior**Otros objetos**

Los otros objetos que incluye BOM son:

- [document](#): el objeto `document` que hemos visto en el DOM
- [navigator](#): nos informa sobre el navegador y el sistema en que se ejecuta
  - `.userAgent`: muestra información sobre el navegador que usamos
  - `.platform`: muestra información sobre la plataforma sobre la que se ejecuta
  - ...
- [screen](#): nos da información sobre la pantalla
  - `.width/.height`: ancho/alto total de la pantalla (resolución)
  - `.availWidth/.availHeight`: igual pero excluyendo la barra del S.O.
  - ...

**EJERCICIO:** obtén todas las propiedades `width/height` y `availWidth/availHeight` del objeto `screen`. Compáralas con las propiedades `innerWidth/innerHeight` y `outerWidth/outerHeight` de `window`.



## DWEC – Javascript Web Cliente.

# Javascript – HTML - Drag And Drop API

Con HTML5 es muy sencillo arrastrar y soltar elementos en una página web. Podemos arrastrar y soltar cualquier nodo DOM (una imagen, un archivo, enlaces, texto seleccionado, ...). Para ello sólo es necesario que ese elemento tenga el atributo `dragable="true"`. Si le ponemos `false` no se podrá arrastrar y si no definimos el atributo podrá o no arrastrarse según el valor predeterminado del navegador (en la mayoría son *dragables* las imágenes, los links y las selecciones de texto).

Al arrastrar y soltar intervienen 2 elementos diferentes:

- el elemento que estamos arrastrando
- el elemento sobre el cual lo soltamos

Para poder realizar la operación `event` tiene una propiedad llamada **`dataTransfer`** que es un objeto en el que almacenamos qué elemento estamos arrastrando (o cualquier otra cosa que queramos) y así cuando se suelte sobre el elemento destino éste último pueda saber quién se le ha soltado.

Los pasos para arrastrar y soltar un elemento son:

1. El elemento debe ser **`draggable`**
2. Capturamos el evento **`dragstart`**. Este evento se produce sobre un elemento cuando comenzamos a arrastrarlo. Deberemos almacenar en el `dataTransfer` quién está siendo arrastrado (si no guardamos nada se guarda automáticamente su `src` si es una imagen o su `href` si es un enlace). Indicaremos el tipo del dato que estamos almacenando (texto plano, HTML, fichero, etc) y su valor. Ej.:

```
img id="imgGoogle"
src="https://upload.wikimedia.org/wikipedia/commons/thumb/5/51/Google.png/320px-Google.png">
<div id="zonaDrop1" class="drop">
  <p>Puedes soltar aquí la imagen</p>
</div>
<div id="zonaDrop2" class="drop">
  <p>Y también aquí</p>
</div>
document.getElementById('imgGoogle').addEventListener('dragstart', (event) => {
  event.dataTransfer.setData('text/plain', event.target.id); // Estamos guardando el texto 'imgGoogle'
})
```

1. Capturamos el evento **`dragover`**. Este evento se produce cada pocas décimas de segundo sobre el elemento sobre el que se está arrastrando algo. Por defecto no se puede soltar un elemento en ningún sitio, así que capturamos este evento para evitar que el navegador haga la acción por defecto e impida que se suelte lo que estamos arrastrando. Ej.:

```
document.getElementById('zonaDrop1').addEventListener('dragover', (event) => {
  event.preventDefault();
})
document.getElementById('zonaDrop2').addEventListener('dragover', (event) => {
  event.preventDefault();
})
```

2. Capturamos el evento **drop**. Este evento se produce sobre el elemento sobre el que se suelta lo que estábamos arrastrando. Lo que haremos es evitar el comportamiento por defecto del navegador (que en caso de imágenes o enlaces es cargarlos en la página), obtener quién se ha soltado a partir del objeto *dataTransfer* y realizar lo que queramos, que normalmente será añadir el objeto arrastrado como hijo del objeto sobre el que se ha hecho el *drop*. Ej.:

```
document.getElementById('zonaDrop1').addEventListener('drop', (event) => {
  event.preventDefault();
  const data=event.dataTransfer.getData("text/plain");           // Obtenemos imgGoogle
  event.target.appendChild(document.getElementById(data));
})
document.getElementById('zonaDrop2').addEventListener('drop', (event) => {
  event.preventDefault();
  const data=event.dataTransfer.getData("text/plain");           // Obtenemos imgGoogle
  event.target.appendChild(document.getElementById(data));
})
```

## Ejemplo:

### HTML:

```
<p draggable="true" class="draggable" id="text-to-move">
  Texto para mover
</p>
<div id="zonaDrop1" class="drop">
  <p>Puedes soltar aquí la imagen</p>
</div>
<div id="zonaDrop2" class="drop">
  <p>Y también aquí</p>
</div>
```

### CSS:

```
.drop {
  border: 1px solid black;
  height: 60px;
}

.draggable {
  background-color: yellow;
}
```

## JavaScript:

```
document.getElementById('zonaDrop1').addEventListener('dragover', (event) => {
  event.preventDefault();
})
document.getElementById('zonaDrop2').addEventListener('dragover', (event) => {
  event.preventDefault();
})

document.getElementById('zonaDrop1').addEventListener('drop', (event) => {
  event.preventDefault();
  const data=event.dataTransfer.getData("text/plain");
  event.target.appendChild(document.getElementById(data));
})
document.getElementById('zonaDrop2').addEventListener('drop', (event) => {
  event.preventDefault();
  const data=event.dataTransfer.getData("text/plain");
  event.target.appendChild(document.getElementById(data));
})
```

Podemos obtener más información de esta API [MDN web docs](#). Y en el ejemplo adjunto como archivo zip.



## DWEC – Javascript Web Cliente.

JavaScript 06 – Eventos.....	1
Introducción.....	1
Escuchar un evento utilizando un escuchador o <i>listener</i> .....	1
Event listeners.....	2
Tipos de eventos .....	3
Eventos de carga .....	3
Eventos de ratón.....	3
Eventos de teclado.....	4
Eventos de toque .....	4
Eventos de formulario.....	4
Los objetos event y this .....	4
event .....	5
this .....	6
Bindeo del objeto this .....	7
Propagación de eventos (bubbling) .....	10
innerHTML y escuchadores de eventos .....	12
Eventos personalizados .....	13

# JavaScript 06 – Eventos

## Introducción

Los eventos permiten detectar acciones que realiza el usuario o cambios que suceden en la página para poder reaccionar como respuesta.

Existen muchos eventos diferentes, se puede ver la lista en [W3schools](#). Nos centraremos en los más comunes.

Cualquier elemento HTML del documento puede tener asociado uno o varios eventos DOM. También podemos asociar otros tipos de eventos, como se puede consultar en <https://developer.mozilla.org/es/docs/Web/Events>

Javascript nos permite ejecutar código cuando se produce un evento (por ejemplo: el evento *click* del ratón) asociando al mismo una función. Hay varias formas de hacerlo.

## Escuchar un evento utilizando un escuchador o *listener*.

La primera manera “estándar” de asociar código a un evento era añadiendo un atributo con el nombre del evento a escuchar (con ‘on’ delante) en el elemento HTML.

Por ejemplo, para ejecutar código al producirse el evento ‘click’ sobre un botón se escribía:

```
<input type="button" id="boton1" onclick="alert('Se ha pulsado');" />
```

Una mejora era llamar a una función que contenía el código:

```
<input type="button" id="boton1" onclick="clicked()" />
```

```
function clicked() {
    alert('Se ha pulsado');
}
```

Esto “ensuciaba” con código la página HTML, por lo que se creó el modelo de registro de eventos tradicional que permitía asociar a un elemento HTML una propiedad con el nombre del evento a escuchar (con ‘on’ delante). En el caso anterior sería:

```
document.getElementById('boton1').onclick = function () {
    alert('Se ha pulsado');
}
...
```

NOTA: hay que tener cuidado porque si se ejecuta el código antes de que se haya creado el botón estaremos asociando la función al evento *click* de un elemento que aún no existe, así que no hará nada.

Para evitarlo, es conveniente poner el código que atiende a los eventos dentro de una función que se ejecute al producirse el evento *load* de la ventana. Este evento se produce cuando se han cargado todos los elementos HTML de la página y se ha creado el árbol DOM. (*También podemos evitarlo si el script es llamado al final del BODY o si el script es de tipo “module”*).

Lo mismo habría que hacer con cualquier código que modifique el árbol DOM. El código correcto sería:

```
window.onload = function() {
    document.getElementById('boton1').onclick = function() {
        alert('Se ha pulsado');
    }
}
```

## Event listeners

La forma recomendada de escuchar un evento es usando el modelo avanzado de registro de eventos del W3C.

Se usa el método `addEventListener` que recibe:

- como primer parámetro el nombre del evento a escuchar (sin ‘on’)
- y como segundo parámetro la función a ejecutar (OJO, sin paréntesis) cuando se produzca el evento:

```
document.getElementById('boton1').addEventListener('click', pulsado);
...
function pulsado() {
    alert('Se ha pulsado');
})
```

Habitualmente se usan funciones anónimas, ya que no necesitan ser llamadas desde fuera del escuchador:

```
document.getElementById('boton1').addEventListener('click', function() {
    alert('Se ha pulsado');
});
```

Si queremos pasar algún parámetro a la función escuchadora, poco habitual, debemos usar funciones anónimas como escuchadores de eventos:

```
<button id="acepto">Aceptar</button>
```

```
window.addEventListener('load', function() {
  document.getElementById('acepto').addEventListener('click', function() {
    alert('Se ha aceptado');
  })
})
```

NOTA: igual que antes, debemos estar seguros de que se ha creado el árbol DOM antes de poner un escuchador, por lo que se recomienda ponerlos siempre dentro de la función asociada al evento `window.onload`. O mejor: `window.addEventListener('load', ...)` como se ve en el ejemplo anterior.

Una ventaja de este método es que podemos poner varios escuchadores para el mismo evento y se ejecutarán todos ellos.

Para eliminar un escuchador se usa el método `removeEventListener`.

```
document.getElementById('acepto').removeEventListener('click', aceptar);
```

NOTA: no se puede quitar un escuchador si hemos usado una función anónima. Para quitarlo debemos usar como escuchador una función con nombre.

## Tipos de eventos

Según qué o dónde se produzca un evento, estos se clasifican en:

### Eventos de carga

Se producen en el objeto `window`:

- **load**: se produce cuando termina de cargarse la página (cuando ya está construido el árbol DOM). Es útil para hacer acciones que requieran que el DOM esté cargado, como modificar la página o poner escuchadores de eventos
- **unload**: al destruirse el documento (ej. cerrar)
- **beforeUnload**: antes de destruirse (podríamos mostrar un mensaje de confirmación)
- **resize**: cuando cambia el tamaño del documento (porque se redimensiona la ventana)

### Eventos de ratón

Los produce el usuario con el ratón:

- **click / dblclick**: cuando se hace *click/doble click* sobre un elemento
- **mousedown / mouseup**: al *pulsar/soltar* cualquier botón del ratón
- **mouseenter / mouseleave**: cuando el puntero del ratón *entra/sale* del elemento (tb. podemos usar *mouseover/mouseout*)
- **mousemove**: se produce continuamente mientras el puntero se *movea* dentro del elemento

NOTA: si hacemos “doble click” sobre un elemento, la secuencia de eventos que se produciría es: *mousedown -> mouseup -> click -> mousedown -> mouseup -> click -> dblclick*

#### EJERCICIO:

- Pon un escuchador al botón 1 de la [página de ejemplo de DOM](#) para que al hacer click se muestre un alert con ‘Click sobre botón 1’
- Pon otro escuchador al mismo botón para que se abra otra ventana nueva (de 200 px de ancho y 100 de alto) con un texto dentro que reze “Nueva ventana emergente”. **Nota:** Comprueba si hay diferencias si se abre la página desde “Live Server” o directamente como archivo local.
- Pon otro *listener* al mismo botón para que al pasar el ratón sobre él se muestre debajo de los botones un párrafo en rojo con la frase “Se va a abrir una ventana nueva”.
- Pon otro escuchador al mismo botón para que cuando salga el cursor del ratón desaparezca el párrafo del apartado anterior.
- Pon un escuchador al botón 2 que desactive el escuchador del primer apartado.

## Eventos de teclado

Los produce el usuario al usar el teclado:

- keydown:** se produce al presionar una tecla y se repite continuamente si la tecla se mantiene pulsada
- keyup:** cuando se deja de presionar la tecla
- keypress:** acción de pulsar y soltar (sólo se produce en las teclas alfanuméricas)

NOTA: el orden de secuencia de los eventos es: *keyDown -> keyPress -> keyUp*

## Eventos de toque

Se producen al usar una pantalla táctil:

- touchstart:** se produce cuando se detecta un toque en la pantalla táctil
- touchend:** cuando se deja de pulsar la pantalla táctil
- touchmove:** cuando un dedo es desplazado a través de la pantalla
- touchcancel:** cuando se interrumpe un evento táctil.

## Eventos de formulario

Se producen en los formularios:

- focus / blur:** al obtener/perder el foco el elemento.
- change:** al perder el foco un `<input>` o `<textarea>` si ha cambiado su contenido, o al cambiar de valor un `<select>` o un `<checkbox>`.
- input:** al cambiar el valor de un `<input>` o `<textarea>`. Se produce cada vez que escribimos una letra en estos elementos.
- select:** al cambiar el valor de un `<select>` o al seleccionar texto de un `<input>` o `<textarea>`.
- submit / reset:** al enviar/recargar un formulario.

## Los objetos event y this

Al producirse un evento, se generan automáticamente en su función manejadora 2 objetos: **this** y **event**.

## event

**event**: es un objeto, y la función escuchadora lo recibe como parámetro. Tiene propiedades y métodos que nos dan información sobre el evento, como:

- **.type**: qué evento se ha producido (click, submit, keyDown, ...)
- **.target**: el elemento donde se produjo el evento. Puede ser *this* o un descendiente de *this*, como se ve en el ejemplo que hay un poco más abajo.
- **.currentTarget**: Identifica el target (objetivo) actual del evento, ya que el evento atraviesa el DOM. Siempre hace referencia al elemento al cual el controlador del evento fue asociado, a diferencia de *event.target*, que identifica el elemento en el que se produjo el evento.

**Ejemplo:** Tenemos un elemento *P* al que le ponemos un escuchador de ‘click’ que dentro tiene un elemento *STRONG*.

Si hacemos *\_click* sobre el elemento *STRONG*: **event.target** valdrá el *STRONG* que es donde hemos hecho click (está dentro de *<p>*), pero tanto para *P* como para *STRONG* **.event.currentTarget** valdrá el elemento *<p>* (que es quien tiene el escuchador que se está ejecutando).

- **.relatedTarget**: en un evento ‘mouseover’: **event.target** es el elemento donde ha entrado el puntero del ratón y **event.relatedTarget** el elemento del que ha salido. En un evento ‘mouseout’: sería al revés.
- **.cancelable**: si el evento puede cancelarse. En caso afirmativo se puede llamar a **event.preventDefault()** para cancelarlo
- **.preventDefault()**: si un evento tiene un escuchador asociado se ejecuta el código de dicho escuchador y después el navegador realiza la acción que correspondería por defecto al evento si no tuviera escuchador.

Por ejemplo: un escuchador del evento *click* sobre un hiperenlace hará que se ejecute su código y después saltará a la página indicada en el *href* del hiperenlace. Este método cancela la acción por defecto del navegador para el evento.

Otro ejemplo de uso de este método: si el evento era el *submit* de un formulario éste no se enviará, o si era un *click* sobre un hiperenlace no se irá a la página indicada en él.

- **.stopPropagation()**: Como por defecto un evento se produce sobre un elemento y todos sus padres, al usar este método se evita esta propagación.

Por ejemplo: si hacemos click en un *<span>* que está en un *<p>* que está en un *<div>* que está en el *BODY*, el evento se va propagando por todos estos elementos y saltarían los escuchadores asociados a todos ellos (si los hubiera). Si algún escuchador llama a este método, el evento no se propagará a los demás elementos padre.

*Un evento, dependiendo del tipo, puede tener más propiedades:*

*eventos de ratón:*

- **.button**: qué botón del ratón se ha pulsado (0: izq, 1: rueda; 2: dcho).
- **.screenX / .screenY**: las coordenadas del ratón respecto a la pantalla.
- **.clientX / .clientY**: las coordenadas del ratón respecto a la ventana cuando se produjo el evento.
- **.pageX / .pageY**: las coordenadas del ratón respecto al documento (si se ha hecho un scroll será el clientX/Y más el scroll).

- **.offsetX / .offsetY**: las coordenadas del ratón respecto al elemento sobre el que se produce el evento.
- **.detail**: si se ha hecho click, doble click o triple click

#### eventos de teclado:

Son los más incompatibles entre diferentes navegadores. En el teclado hay teclas normales y especiales (Alt, Ctrl, Shift, Enter, Tab, flechas, Supr, ...). En la información del teclado hay que distinguir entre el código del carácter pulsado (e=101, E=69, €=8364) y el código de la tecla pulsada (para los 3 caracteres es el 69 ya que se pulsa la misma tecla). Las principales propiedades de *event* son:

- **.key**: devuelve el nombre de la tecla pulsada
- **.which**: devuelve el código de la tecla pulsada
- **.keyCode / .charCode**: código de la tecla pulsada y del carácter pulsado (según navegadores)
- **.shiftKey / .ctrlKey / .altKey / .metaKey**: si está o no pulsada la tecla SHIFT / CTRL / ALT / META. Esta propiedad también la tienen los eventos de ratón

**NOTA:** a la hora de saber qué tecla ha pulsado el usuario es conveniente tener en cuenta:

- Para saber qué carácter se ha pulsado lo mejor es usar la propiedad *key* o *charCode* de *keyPress*, pero varía entre navegadores.
- Para saber la tecla especial pulsada, mejor usar el *key* o el *keyCode* de *keyUp*.
- Hay que capturar sólo lo que sea necesario, se producen muchos eventos de teclado.
- Para obtener el carácter a partir del código, se aconseja utilizar: `String.fromCharCode(n1, n2, ...)`
- Lo mejor para familiarizarse con los diferentes eventos es consultar los [ejemplos de w3schools](#).

**EJERCICIO A:** Pon un escuchador al BODY de la [página de ejemplo](#) para que al mover el ratón en cualquier punto de la ventana del navegador, se muestre en los distintos DIV la posición del puntero respecto del navegador y respecto de la página.

**EJERCICIO B:** Pon un listener al BODY de la [página de ejemplo](#) para que al pulsar cualquier tecla nos muestre en un párrafo el *key* y el *keyCode* de la tecla pulsada. Pruébalo con diferentes teclas.

## this

**this**: siempre hace referencia al elemento que contiene el código en donde se encuentra la variable *this*. En el caso de una función escuchadora será el elemento que tiene el escuchador que ha recibido el evento.

**Ojo!**: No usar funciones flecha. Más información en el documento JavaScript - Anexo - “Uso de this en contexto”.

Veamos un ejemplo:

```
<!DOCTYPE html>
<html lang="es">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Ejemplo 1 DOM</title>
  </head>
  <body>
```

```

<h1 class="important">Ejemplo de eventos con this</h1>

<form action="#">
  <label for="input1">
    Dato 1 <input id="input1" type="text" size="20" /><br />
  </label>
  <label for="input2">
    Dato 2 <input id="input2" type="text" size="20" /><br />
  </label>
  <label for="input3">Dato 3
    <input id="input3" type="text" size="20" /><br />
  </label>
</form>
<script src="js/this.js"></script>
</body>
</html>

```

```

const arrayInputs=Array.from(document.getElementsByTagName('input'));

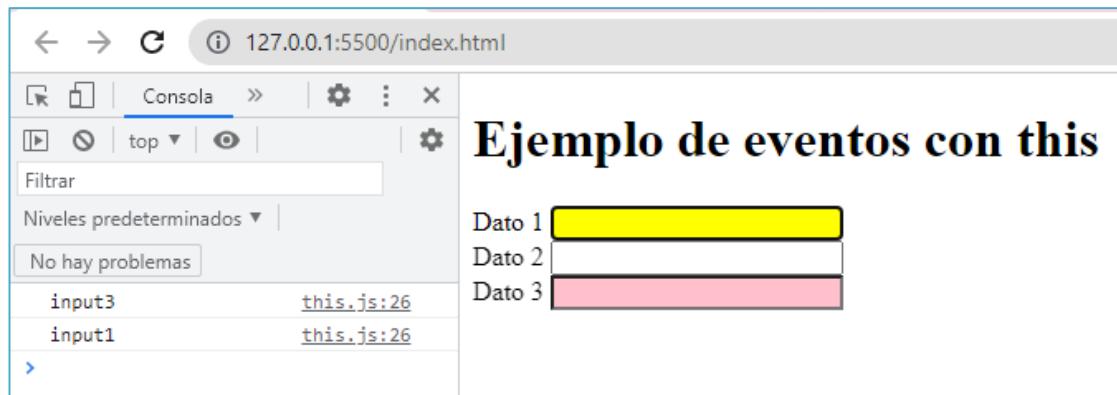
function pintar(event){
  console.log(event.target.id);
  this.style.backgroundColor='yellow';
}

function pintarRosa(event){
  this.style.backgroundColor='pink';
}

arrayInputs.forEach(element => {
  element.addEventListener('focus', pintar); //Al hacer clic los pone en amarillo
  element.addEventListener('blur', pintarRosa); // Al salir los deja rosa (para siempre)
});-

```

Ejemplo de ejecución:



## Bindeo del objeto this

El "bind" en JavaScript se utiliza para cambiar el valor de this en una función. Esto es útil en situaciones donde el valor predeterminado de this no es el que se desea. El método bind() crea una nueva función que, cuando se llama,

tiene su propio valor de `this` especificado, con una secuencia dada de argumentos precediendo a cualquier otro proporcionado cuando la nueva función es llamada.

Prueba el siguiente código:

```
let persona = {
    nombre: "Juan",
    edad: 25,
    decirHola: function () {
        console.log("Hola, soy " + this.nombre);
    },
};

persona.decirHola(); // Imprimirá: Hola, soy Juan

let otraPersona = { nombre: "María", edad: 33 };

let funcionSaludo = persona.decirHola.bind(otraPersona);
funcionSaludo(); // Imprimirá: Hola, soy María

persona.decirHola.bind(otraPersona); // No hace nada porque aunque llamamos a bind,
                                    // no hacemos nada con la función resultante
persona.decirHola(); // Seguirá imprimiendo: Hola, soy Juan

// Para cambiar permanentemente el contexto de la función en persona
persona.decirHola = persona.decirHola.bind(otraPersona);
persona.decirHola(); // Ahora ya siempre imprimirá: Hola, soy María
```

### **Bind en el caso de eventos:**

En ocasiones no queremos que `this` sea el elemento sobre quien se produce el evento, si no que queremos conservar el valor que tenía `this` antes de entrar a la función escuchadora.

Por ejemplo: Si la función escuchadora es un método de una clase, en `this` tenemos el objeto de la clase sobre el que estamos actuando, pero al entrar en la función perdemos esa referencia.

El método `.bind()` nos permite pasarle a una función el valor que queremos darle a la variable `this` dentro de dicha función. **Bind** podemos traducirlo como **forzar**.

Por defecto a una función escuchadora de eventos se le bindea el valor de `event.currentTarget`. Si queremos que tenga otro valor se lo indicamos con `.bind()`:

```
function probar1(evento){
    console.log(this); //elemento prueba
    console.log(this.innerText); // texto de prueba
    console.log(evento); //puntero a evento
    console.log(evento.target); //elemento prueba
    console.log(evento.currentTarget); // elemento prueba
}

function probar2(evento){
    console.log(this); // objeto Window
```

```

        console.log(evento); // puntero a evento
        console.log(evento.target); //elemento prueba
        console.log(evento.currentTarget); //elemento prueba
    }

function probar3(var1, var2, var3, evento){
    // Hay que pasarle en la llamada 4 argumentos para que no haya error
    console.log(this); // objeto Window
    console.log(evento); // puntero a evento
    console.log(evento.target); //elemento prueba
    console.log(evento.currentTarget); //elemento prueba
    console.log(var1); // 4
    console.log(var2); // pepe
    console.log(var3); // 7
}

function probar4(var1, evento){
    console.log(var1); // puntero a evento
    console.log(this); // "argumento"
    console.log(evento); // undefined
    console.log(evento.target); //error
    console.log(evento.currentTarget); // error
}

// id="prueba" es un elemento H1, h2, P, o similar.
document.getElementById("prueba").addEventListener('click', probar1);
document.getElementById("prueba").addEventListener('click', probar2.bind(this));
document.getElementById("prueba").addEventListener('click', probar3.bind(this, 4, "pepe",7));
document.getElementById("prueba").addEventListener('click', probar4.bind("argumento"));

```

Más ejemplos:

```
document.getElementById('acepto').addEventListener('click', aceptar.bind(variable));
```

En este ejemplo, el valor de *this* dentro de la función *aceptar* será *variable*.

En el ejemplo que habíamos comentado de un escuchador dentro de una clase, para mantener el valor que trae *this* y que haga referencia al objeto sobre el que estamos actuando haríamos:

```
document.getElementById('acepto').addEventListener('click', aceptar.bind(this));
```

por lo que el valor de *this* dentro de la función *aceptar* será el mismo que tenía fuera, es decir, el objeto.

Podemos *bindear*, es decir, pasarle a la función escuchadora más variables, declarándolas como parámetros de *bind*. El primer parámetro será el valor de *this*, y los demás serán parámetros que recibirá la función antes de recibir el parámetro *event*, que será el último. Por ejemplo:

```

document.getElementById('acepto').removeEventListener('click', aceptar.bind(var1, var2,
var3));
...
function aceptar(param1, param2, event) {
    // Aquí dentro tendremos los valores
}

```

```
// this = var1
// param1 = var2
// param2 = var3
// event es el objeto con la información del evento producido
}
```

Ejercicio: Crear una tabla con varias filas y columnas que tenga dentro de cada celda los valores de un array. Según se haga clic en cada celda, que se muestre en consola el contenido de dicha celda.

Ejercicio: Modifica el ejercicio anterior de tal forma que las celdas aparecerán vacías, y al hacer clic sobre cada celda cambie el contenido de una variable global numérica autoincremental.

Ejercicio: Crear una página web con dos cuadros de texto (`<input type="text">`) y un botón (`<button>`). Cuando se haga clic en el botón, debería cambiar el color de fondo de los cuadros de texto. Utiliza bind para vincular el contexto apropiado.

Ejercicio: Modifica el ejercicio anterior de tal forma que le pase un texto com parámetro a cada input en la llamada a la función escuchadora.

## Propagación de eventos (bubbling)

Normalmente en una página web los elementos HTML se solapan unos con otros, por ejemplo: un `<span>` está en un `<p>` que está en un `<div>` que está en el `<body>`. Si ponemos un escuchador del evento `click` a todos ellos se ejecutarán todos ellos, pero ¿en qué orden?

Pues el W3C estableció dos modelos conocidos como **fase de captura** y la segunda como **fase de burbujeo**.

- En el modelo de burbujeo: se disparan los eventos de dentro hacia fuera.
- En el modelo de captura: primero se disparan los eventos de fuera hacia dentro (primero el `<body>`) y al llegar al más interno (el `<spam>`)

Cuando ponemos un escuchador con `addEventListener` el tercer parámetro indica el modelo a utilizar:

- **false** (valor por defecto): en fase de burbujeo
- **true**: en fase de captura

Ejemplo:

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
    <!-- <link rel="stylesheet" href="estilos.css"> -->
    <style>
        div{
            display: flex;
            justify-content: flex-end;
        }
    </style>
</head>
<body>
    <div>
        <input type="text" value="Captura" />
        <input type="text" value="Burbujeo" />
    </div>
</body>
</html>
```

```

        align-items: flex-end;
    }
</style>
</head>
<body>
    <div id="divVerde" style="background-color: green; width: 150px; height: 150px;">
        <div id="divRojo" style="background-color: red; width: 100px; height: 100px;">
            <div id="divAzul" style="background-color: blue; width: 50px; height: 50px;"></div>
        </div>
    </div>
    <p id="info"></p>
    <script src="js/app.js"></script>
</body>
</html>

```

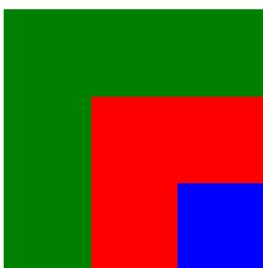
```

let divClick = function(event) {
// eventPhase: 1 -> capture, 2 -> target (objetivo), 3 -> bubble
    document.getElementById('info').innerHTML+="Has pulsado: " + this.id + ". Fase: " +
event.eventPhase + ", this: " + this.id + ", event.target: " + event.target.id + ",
event.currentTarget: " + event.currentTarget.id + "<br>";
};

let divVerde = document.getElementById("divVerde");
let divRojo = document.getElementById("divRojo");
let divAzul = document.getElementById("divAzul");
divVerde.addEventListener('click', divClick);
divRojo.addEventListener('click', divClick);
divAzul.addEventListener('click', divClick);

```

Veremos algo similar a:



### CASO A - Tercer parámetro por defecto (*false*)

Haciendo clic en el div Azul: veremos un resultado así:

```

Has pulsado: divAzul. Fase: 2, this: divAzul, event.target: divAzul, event.currentTarget: divAzul
Has pulsado: divRojo. Fase: 3, this: divRojo, event.target: divAzul, event.currentTarget: divRojo
Has pulsado: divVerde. Fase: 3, this: divVerde, event.target: divAzul, event.currentTarget: divVerde

```

Primero responde Azul en fase 2 (objetivo o destino) y luego responden Rojo y Verde en fase 3 (burbujeo).

Observa que **event.target** siempre es Azul, mientras que **event.currentTarget** va cambiando.

## eventPhase:

La propiedad **eventPhase** es una propiedad de los objetos de eventos en JavaScript que indica la fase en la que se encuentra el evento. Los eventos en JavaScript pasan por tres fases diferentes durante su propagación a través del DOM (Modelo de Objetos de Documento): la fase de captura, la fase de destino y la fase de burbujeo.

La propiedad **eventPhase** tiene tres valores posibles:

1. Event.CAPTURING\_PHASE (1): Esta es la fase de captura. Durante esta fase, el evento desciende por la jerarquía de elementos desde el ancestro más externo hasta el objetivo del evento.
2. Event.AT\_TARGET (2): Esta es la fase de destino. Durante esta fase, el evento llega al objetivo específico al que se dirige.
3. Event.BUBBLING\_PHASE (3): Esta es la fase de burbujeo. Durante esta fase, el evento asciende desde el objetivo del evento hasta el ancestro más externo.

Puedes utilizar la propiedad **eventPhase** en un controlador de eventos para determinar en qué fase se encuentra el evento.

## CASO B - Tercer parámetro con valor *true*

Sin embargo, si al método `.addEventListener` le pasamos un tercer parámetro con el valor *true*, el comportamiento será el contrario, lo que se conoce como *captura*. Y el primer escuchador que se ejecutará es el del `<body>` y el último el del `<span>`

Probando a añadir ese parámetro a los escuchadores del ejemplo anterior con los divs de colores:

```
divVerde.addEventListener('click', divClick, true );
divRojo.addEventListener('click', divClick, true);
divAzul.addEventListener('click', divClick, true );
```

Volviendo a hacer clic en el div Azul: veremos un resultado así:

```
Has pulsado: divVerde. Fase: 1, this: divVerde, event.target: divAzul, event.currentTarget: divVerde
Has pulsado: divRojo. Fase: 1, this: divRojo, event.target: divAzul, event.currentTarget: divRojo
Has pulsado: divAzul. Fase: 2, this: divAzul, event.target: divAzul, event.currentTarget: divAzul
```

Observamos que primero responden Verde y Rojo en fase 1 (captura), y por último Azul en fase 2 (objetivo).

En cualquier momento podemos evitar que se siga propagando el evento ejecutando el método `.stopPropagation()` en el código de cualquiera de los escuchadores.

Se pueden ver las distintas fases de un evento en la página [domevents.dev](http://domevents.dev).

## innerHTML y escuchadores de eventos

Si cambiamos la propiedad *innerHTML* de un elemento del árbol DOM, todos sus escuchadores de eventos desaparecen ya que es como si se volviera a crear ese elemento (y los escuchadores deben ponerse después de crearse).

Por ejemplo: tenemos una tabla de datos y queremos que al hacer doble click en cada fila se muestre su id. La función que añade una nueva fila podría ser:

```
function renderNewRow(data) {
  let miTabla = document.getElementById('tabla-datos');
  let nuevaFila = `<tr id="${data.id}"><td>${data.dato1}</td><td>${data.dato2}...</td></tr>`;
```

```

miTabla.innerHTML += nuevaFila;
document.getElementById(data.id).addEventListener('dblclick', event => alert('Id: ' +
event.target.id));
}

```

Sin embargo, esto sólo funcionaría para la última fila añadida ya que la línea `miTabla.innerHTML += nuevaFila` equivale a `miTabla.innerHTML = miTabla.innerHTML + nuevaFila`. Por tanto, estamos asignando a `miTabla` un código HTML que ya no contiene escuchadores, excepto el de `nuevaFila` que lo ponemos después de hacer la asignación.

La forma correcta de hacerlo sería:

```

function renderNewRow(data) {
let miTabla = document.getElementById('tabla-datos');
let nuevaFila = document.createElement('tr');
nuevaFila.id = data.id;
nuevaFila.innerHTML = `<td>${data.dato1}</td><td>${data.dato2}...</td>`;
nuevaFila.addEventListener('dblclick', event => alert('Id: ' + event.target.id) );
miTabla.appendChild(nuevaFila);
}

```

De esta forma además mejoramos el rendimiento, ya que el navegador sólo tiene que renderizar el nodo correspondiente a la `nuevaFila`.

Si lo hacemos como estaba al principio se deben volver a crear y a renderizar todas las filas de la tabla (con todo lo que hay dentro de `miTabla`).

## Eventos personalizados

También podemos, mediante código, lanzar manualmente cualquier evento sobre un elemento con el método `dispatchEvent()`, e incluso crear eventos personalizados. Por ejemplo:

```

const event = new Event('build');

// Listen for the event.
elem.addEventListener('build', (e) => { /* ... */ }, false);

// Dispatch the event.
elem.dispatchEvent(event);

```

Incluso podemos añadir datos al objeto `event` si creamos el evento con `new CustomEvent()`.

Más información en la [página de MDN](#).



## DWEC – Javascript Web Cliente.

JavaScript 07 – Objetos nativos .....	1
Introducción.....	1
Funciones globales.....	1
Objetos nativos del lenguaje.....	2
Objeto Math .....	3
Objeto Date.....	4

# JavaScript 07 – Objetos nativos

## Introducción

En este tema vamos a ver:

- Las funciones globales de Javascript, muchas de las cuales ya hemos visto como *Number()* o *String()*.
- Los objetos nativos que incorpora Javascript y que nos facilitarán el trabajo proporcionándonos métodos y propiedades útiles para no tener que “reinventar la rueda” en nuestras aplicaciones.
- Uno de ellos está el objeto **RegExp** que nos permite trabajar con **expresiones regulares** (son iguales que en otros lenguajes) que nos serán de gran ayuda, sobre todo a la hora de validar formularios y que por eso veremos en la siguiente unidad.

## Funciones globales

- *parseInt(valor)*: devuelve el valor pasado como parámetro convertido a entero o *NaN* si no es posible la conversión. Este método es mucho más permisivo que *Number* y convierte cualquier cosa que comience por un número (si encuentra un carácter no numérico detiene la conversión y devuelve lo convertido hasta el momento). Ejemplos:

```
console.log( parseInt(3.84) )           // imprime 3 (ignora los decimales)
console.log( parseInt('3.84') )          // imprime 3
console.log( parseInt('28manzanas') )    // imprime 28
console.log( parseInt('manzanas28') )    // imprime NaN
```

- *parseFloat(valor)*: igual, pero devuelve un número decimal. Ejemplos:

```
console.log( parseFloat(3.84) )           // imprime 3.84
console.log( parseFloat('3.84') )          // imprime 3.84
console.log( parseFloat('3,84') )          // imprime 3 (la coma no es un carácter numérico)
console.log( parseFloat('28manzanas') )    // imprime 28
console.log( parseFloat('manzanas28') )    // imprime NaN
```

- `Number(valor)`: convierte el valor a un número. Es como `parseFloat` pero más estricto. Si no puede convertir todo el valor, devuelve `NaN`. Ejemplos:

```
console.log( Number(3.84) )           // imprime 3.84
console.log( Number('3.84') )          // imprime 3.84
console.log( Number('3,84') )          // imprime NaN (La coma no es un carácter numérico)
console.log( Number('28manzanas') )    // imprime NaN
console.log( Number('manzanas28') )    // imprime NaN
```

- `String(valor)`: convierte el valor pasado en una cadena de texto. Si le pasamos un objeto llama al método `toString()` del objeto. Ejemplos:

```
console.log( String(3.84) )           // imprime '3.84'
console.log( String([24, 3, 12]) )      // imprime '24,3,12'
console.log( {nombre: 'Marta', edad: 26} ) // imprime "[object Object]"
```

- `Boolean(valor)`: convierte el valor pasado a un booleano. Sería el resultado de tenerlo como condición en un `if`. Muchas veces en vez de usar esto usamos la doble negación `!!` que da el mismo resultado. Ejemplos:

```
console.log( Boolean('Hola') )         // Equivaldría a !!'Hola'. Imprime true
console.log( Boolean(0) )               // Equivaldría a !!0. Imprime false
```

- `isNaN(valor)`: devuelve `true` si lo pasado NO es un número o no puede convertirse en un número. Ejemplos:

```
console.log( isNaN(3.84) )            // imprime false
console.log( isNaN('3.84') )          // imprime false
console.log( isNaN('3,84') )          // imprime true (La coma no es un carácter numérico)
console.log( isNaN('28manzanas') )     // imprime true
console.log( isNaN('manzanas28') )     // imprime true
```

- `isFinite(valor)`: devuelve `false` si es número pasado es infinito (o demasiado grande)

```
console.log( isFinite(3.84) )          // imprime true
console.log( isFinite(3.84 / 0) )        // imprime false
```

- `encodeURI(string) / decodeURI(string)`: transforma la cadena pasada a una URL codificada válida, transformando los caracteres especiales que contenga, excepto `,` `/` `:` `@` `&` `=` `+` `$` `#`. Debemos usarla siempre que vayamos a pasar una URL. Ejemplo:
  - Decoded: “`http://domain.com?val=1 2 3&val2=r+y%6`”
  - Encoded: “`http://domain.com?val=1%202%203&val2=r+y%256`”
- `encodeURIComponent(string) / decodeURIComponent(string)`: transforma también los caracteres que no transforma la anterior. Debemos usarla para codificar parámetros, pero no una URL entera. Ejemplo:
  - Decoded: “`http://domain.com?val=1 2 3&val2=r+y%6`”
  - Encoded: “`http%3A%2F%2Fdomain.com%3Fval%3D1%202%203%26val2%3Dr%2By%256`”

## Objetos nativos del lenguaje

En Javascript casi todo son objetos. Ya hemos visto diferentes objetos:

- `window`

- screen
- navigator
- location
- history
- document

Los 5 primeros se corresponden al modelo de objetos del navegador y *document* se corresponde al modelo de objetos del documento. Todos nos permiten interactuar con el navegador para realizar distintas acciones.

También tenemos los tipos de objetos nativos, que no dependen del navegador. Son:

- Number
- String
- Boolean
- Array
- Function
- Object
- Math
- Date
- RegExp

Además de los tipos primitivos de **número**, **cadena**, **booleano**, **undefined** y **null**, Javascript tiene todos los objetos indicados. Como ya hemos visto, se puede crear un número usando su tipo primitivo (`let num = 5`) o su objeto (`let num = new Number(5)`) pero es mucho más eficiente usar los tipos primitivos. Pero aunque lo creemos usando el tipo de dato primitivo se considera un objeto y tenemos acceso a todas sus propiedades y métodos. Ejemplo: `num.toFixed(2)`

Ya hemos visto las principales propiedades y métodos de [Number](#), [String](#), [Boolean](#) y [Array](#) y aquí vamos a ver los de **Math** y **Date** y en el apartado de validar formularios, las de **RegExp**.

## Objeto Math

Proporciona constantes y métodos para trabajar con valores numéricicos:

- constantes: `.PI` (número pi), `.E` (número de Euler), `.LN2` (algoritmo natural en base 2), `.LN10` (logaritmo natural en base 10), `.LOG2E` (logaritmo de E en base 2), `.LOG10E` (logaritmo de E en base 10), `.SQRT2` (raíz cuadrada de 2), `.SQRT1_2` (raíz cuadrada de 1/2). Ejemplos:

```
console.log( Math.PI )           // imprime 3.141592653589793
console.log( Math.SQRT2 )         // imprime 1.4142135623730951
```

- `Math.round(x)`: redondea x al entero más cercano
- `Math.floor(x)`: redondea x hacia abajo (5.99 → 5. Quita la parte decimal)
- `Math.ceil(x)`: redondea x hacia arriba (5.01 → 6)
- `Math.min(x1,x2,...)`: devuelve el número más bajo de los argumentos que se le pasan.
- `Math.max(x1,x2,...)`: devuelve el número más alto de los argumentos que se le pasan.
- `Math.pow(x, y)`: devuelve  $x^y$  (x elevado a y).
- `Math.abs(x)`: devuelve el valor absoluto de x.
- `Math.random()`: devuelve un número decimal aleatorio entre 0 (incluido) y 1 (no incluido).

Si queremos un número entre otros rangos haremos: `Math.random() * (max - min) + min`  
 O si lo queremos sin decimales, haremos: `Math.round(Math.random() * (max - min) + min)`

- `Math.cos(x)`: devuelve el coseno de x (en radianes).
- `Math.sin(x)`: devuelve el seno de x.
- `Math.tan(x)`: devuelve la tangente de x.
- `Math.sqrt(x)`: devuelve la raíz cuadrada de x

Ejemplos:

```
console.log( Math.round(3.14) )      // imprime 3
console.log( Math.round(3.84) )      // imprime 4
console.log( Math.floor(3.84) )     // imprime 3
console.log( Math.ceil(3.14) )      // imprime 4
console.log( Math.sqrt(2) )         // imprime 1.4142135623730951
```

## Objeto Date

Es la clase que usaremos siempre que vayamos a trabajar con fechas. Al crear una instancia de la clase le pasamos la fecha que queremos crear o lo dejamos en blanco para que nos cree la fecha actual. Si le pasamos la fecha podemos pasarle:

- milisegundos, desde la fecha EPOCH
- cadena de fecha
- valor para año, mes (entre 0 y 11), día, hora, minutos, segundos, milisegundos

Ejemplos:

```
let date1=new Date()      // Mon Jul 30 2018 12:44:07 GMT+0200 (CEST) (es cuando he ejecutado la instrucción)
let date7=new Date(1532908000000)    // Mon Jul 30 2018 00:00:00 GMT+0200 (CEST) (miliseg. desde 1/1/1070)
let date2=new Date('2018-07-30')    // Mon Jul 30 2018 02:00:00 GMT+0200 (CEST) (La fecha pasada a las 0h. GMT)
let date3=new Date('2018-07-30 05:30') // Mon Jul 30 2018 05:30:00 GMT+0200 (CEST) (La fecha pasada a las 05:30h. Local)
let date6=new Date('07-30-2018')    // Mon Jul 30 2018 00:00:00 GMT+0200 (CEST) (OJO: formato MM-DD-AAAA)
let date7=new Date('30-Jul-2018')    // Mon Jul 30 2018 00:00:00 GMT+0200 (CEST) (tb. podemos poner 'Julio')
let date4=new Date(2018,7,30)       // Thu Ago 30 2018 00:00:00 GMT+0200 (CEST) (OJO: 0->Ene, 1->Feb... y a las 0h. Local)
let date5=new Date(2018,7,30,5,30)  // Thu Ago 30 2018 05:30:00 GMT+0200 (CEST) (OJO: 0->Ene, 1->Feb, ...)
```

**EJERCICIO:** Crea en la consola dos variables fecNac1 y fecNac2 que contengan tu fecha de nacimiento. La primera la creas pasando una cadena y la segunda pasando año, mes y día.

Cuando ponemos la fecha como texto, como separador de las fechas podemos usar `,` o `espacio`.

Como separador de las horas debemos usar `:`

Cuando ponemos la fecha como parámetros numéricos (separados por `,`) podemos poner valores fuera de rango que se sumarán al valor anterior. Por ejemplo:

```
let date=new Date(2018,7,41)    // Mon Sep 10 2018 00:00:00 GMT+0200 (CEST) -> 41=31Ago+10
let date=new Date(2018,7,0)     // Tue Jul 31 2018 00:00:00 GMT+0200 (CEST) -> 0=0Ago=31Jul (el anterior)
let date=new Date(2018,7,-1)    // Mon Jul 30 2018 00:00:00 GMT+0200 (CEST) -> -1=0Ago-1=31Jul-1=30Jul
```

OJO con el rango de los meses que empieza en 0->Ene, 1->Feb,...,11->Dic

Tenemos métodos **getter** y **setter** para obtener o cambiar los valores de una fecha:

- **fullYear**: permite ver (*get*) y cambiar (*set*) el año de la fecha:

```
let fecha=new Date('2018-07-30')      // Mon Jul 30 2018 02:00:00 GMT+0200 (CEST)
console.log( fecha.getFullYear() )    // imprime 2018
fecha.setFullYear(2019)             // Tue Jul 30 2019 02:00:00 GMT+0200 (CEST)
```

- **month**: devuelve/cambia el número de mes, pero recuerda que 0->Ene,...,11->Dic

```
let fecha=new Date('2018-07-30')      // Mon Jul 30 2018 02:00:00 GMT+0200 (CEST)
console.log( fecha.getMonth() )        // imprime 6
fecha.setMonth(8)                   // Mon Sep 30 2019 02:00:00 GMT+0200 (CEST)
```

- **date**: devuelve/cambia el número de día:

```
let fecha=new Date('2018-07-30')      // Mon Jul 30 2018 02:00:00 GMT+0200 (CEST)
console.log( fecha.getDate() )        // imprime 30
fecha.setDate(-2)                  // Thu Jun 28 2018 02:00:00 GMT+0200 (CEST)
```

- **day**: devuelve el número de día de la semana (0->Dom, 1->Lun, ..., 6->Sáb). Este método NO tiene *setter*:

```
let fecha=new Date('2018-07-30')      // Mon Jul 30 2018 02:00:00 GMT+0200 (CEST)
console.log( fecha.getDay() )         // imprime 1
```

- **hours, minutes, seconds, milliseconds**, : devuelve/cambia el número de hora, minuto, segundo o milisegundo, respectivamente.
- **time**: devuelve/cambia el número de milisegundos desde Epoch (1/1/1970 00:00:00 GMT):

```
let fecha=new Date('2018-07-30')      // Mon Jul 30 2018 02:00:00 GMT+0200 (CEST)
console.log( fecha.getTime() )        // imprime 1532908800000
fecha.setTime(1000*60*60*24*25)     // Fri Jan 02 1970 01:00:00 GMT+0100 (CET) (Le hemos
añadido 25 días a Epoch)
```

**EJERCICIO:** Realiza en la consola los siguientes ejercicios (usa las variables que creaste antes)

- muestra el día de la semana en que naciste
- modifica fecNac1 para que contenga la fecha de tu cumpleaños de este año (cambia sólo el año)
- muestra el día de la semana de tu cumpleaños de este año
- calcula el nº de días que han pasado desde que naciste hasta hoy

Para mostrar la fecha tenemos varios métodos diferentes:

- **.toString()**: “Mon Jul 30 2018 02:00:00 GMT+0200 (CEST)”
- **.toUTCString()**: “Mon, 30 Jul 2018 00:00:00 GMT”
- **.toDateString()**: “Mon, 30 Jul 2018”
- **.toTimeString()**: “02:00:00 GMT+0200 (hora de verano de Europa central)”
- **.toISOString()**: “2018-07-30T00:00:00.000Z”
- **.toLocaleString()**: “30/7/2018 2:00:00”
- **.toLocaleDateString()**: “30/7/2018”
- **.toLocaleTimeString()**: “2:00:00”

**EJERCICIO:** muestra en distintos formatos la fecha y la hora de hoy

**NOTA:** recuerda que las fechas son objetos y que se copian y se pasan como parámetro por referencia:

```
let fecha=new Date('2018-07-30')      // Mon Jul 30 2018 02:00:00 GMT+0200 (CEST)
let otraFecha=fecha
otraFecha.setDate(28)                  // Thu Jun 28 2018 02:00:00 GMT+0200 (CEST)
console.log( fecha.getDate() )         // imprime 28 porque fecha y otraFecha son el mismo
objeto
```

Una forma sencilla de copiar una fecha es crear una nueva pasándole la que queremos copiar:

```
let fecha=new Date('2018-07-30')      // Mon Jul 30 2018 02:00:00 GMT+0200 (CEST)
let otraFecha=new Date(fecha)
otraFecha.setDate(28)                  // Thu Jun 28 2018 02:00:00 GMT+0200 (CEST)
console.log( fecha.getDate() )         // imprime 30
```

En realidad lo que le estamos pasando es el tiempo Epoch de la fecha (es como hacer otraFecha=new Date(fecha.getTime()))

**NOTA:** la comparación entre fechas funciona correctamente con los operadores >, >=, < y <=

Pero NO funciona con ==, ===, != y !== ya que compara los objetos y ve que son objetos diferentes.

Si queremos saber si 2 fechas son iguales (siendo diferentes objetos):

- el código que pondremos NO es fecha1 === fecha2
- pondremos fecha1.getTime() === fecha2.getTime().

EJERCICIO: comprueba si es mayor tu fecha de nacimiento o el 1 de enero de este año.

Podemos probar los distintos métodos de las fechas en la página de [w3schools](#).



## DWEC – Javascript Web Cliente.

JavaScript 08 – Validación de formularios .....	1
Introducción.....	1
Validación del navegador incorporada en HTML5 .....	2
Validación mediante la API de validación de formularios.....	3
Librerías Javascript para validación.....	10
Expresiones regulares .....	10

# JavaScript 08 – Validación de formularios

## Introducción

En este tema vamos a ver cómo realizar una de las acciones principales de Javascript que es la validación de formularios en el lado cliente.

Se trata de una verificación útil porque evita enviar datos al servidor que sabemos que no son válidos, pero NUNCA puede sustituir a la validación en el lado servidor, ya que en el lado cliente otro usuario puede manipular el código desde la consola y hacer que se salten las validaciones que le pongamos.

Antes de enviar datos al servidor, es importante asegurarse de que se completan todos los controles de formulario requeridos, y en el formato correcto. Esto se denomina **validación de formulario en el lado del cliente** y ayuda a garantizar que los datos que se envían coinciden con los requisitos establecidos en los diversos controles de formulario.

La validación en el lado del cliente es una verificación inicial y una característica importante para garantizar una buena experiencia de usuario mediante la detección de datos no válidos en el lado del cliente que el usuario puede corregir de inmediato. Si el servidor recibe datos y, a continuación, los rechaza, se produce un retraso considerable en la comunicación entre el servidor y el cliente que insta al usuario a corregir sus datos.

Si observamos una web bien hecha que incluya un formulario de registro, veremos que proporciona mensajes o comentarios cuando no se introducen datos en el formato que se espera. Se verán mensajes del tipo:

- «Este campo es obligatorio» (No se puede dejar este campo en blanco).
- «Introduzca su número de teléfono en el formato xxx-xxxx» (Se requiere un formato de datos específico para que se considere válido).
- «Introduzca una dirección de correo electrónico válida» (los datos que introdujiste no están en el formato correcto).
- «Su contraseña debe tener entre 8 y 30 caracteres y contener una letra mayúscula, un símbolo y un número». (Se requiere un formato de datos muy específico para tus datos).

Esto se llama **validación de formulario**: Cuando al introducir datos, el navegador y/o el servidor web verifican que estén en el formato correcto y dentro de las restricciones establecidas por la aplicación. La validación realizada en el navegador se denomina validación **en el lado del cliente**, mientras que la validación realizada en el servidor se denomina validación **en el lado del servidor**. Aquí nos centraremos en la validación en el lado del cliente.

Si la información está en el formato correcto, la aplicación permite que los datos se envíen al servidor y (en general) que se guarden en una base de datos. Si la información no está en el formato correcto, da al usuario un mensaje de error que explica lo que debe corregir y le permite volver a intentarlo.

Razones principales para aplicar validación:

- **Queremos obtener los datos correctos en el formato correcto.** Nuestras aplicaciones no funcionarán correctamente si los datos de nuestros usuarios se almacenan en el formato incorrecto, son incorrectos o se omiten por completo.
- **Queremos proteger los datos de nuestros usuarios.** Obligar a nuestros usuarios a introducir contraseñas seguras facilita proteger la información de su cuenta.
- **Queremos protegernos a nosotros mismo.** Hay muchas formas en que los usuarios maliciosos puedan usar mal los formularios desprotegidos y dañar la aplicación (consulta el anexo [Seguridad de sitios web](#)).

Básicamente tenemos 2 maneras de validar un formulario en el lado cliente:

1. Usar la validación incorporada en HTML5 y dejar que sea el navegador quien se encargue de todo.
2. Realizar nosotros la validación mediante Javascript. Lo que puede incluir algunas o todas de las siguientes técnicas:
  - Hacer una API para validación de formularios
  - Utilizar librerías de validación
  - Usar expresiones regulares

La ventaja de la primera opción es que no tenemos que escribir código, sino simplemente poner unos atributos a los INPUT que indiquen qué se ha de validar. La principal desventaja es que no tenemos ningún control sobre el proceso, lo que provocará cosas como:

- El navegador valida campo a campo: cuando encuentra un error en un campo lo muestra y hasta que no se soluciona no valida el siguiente lo que hace que el proceso sea molesto para el usuario que no ve todo lo que hay mal de una vez.
- Los mensajes son los predeterminados del navegador y en ocasiones pueden no ser muy claros para el usuario.
- Los mensajes se muestran en el idioma en que está configurado el navegador, no en el de nuestra página.

## Validación del navegador incorporada en HTML5

Funciona añadiendo atributos a los campos del formulario que queremos validar. Los más usados son:

- **required:** indica que el campo es obligatorio. La validación fallará si no hay nada escrito en el input. En el caso de un grupo de *radiobuttons* se pone sobre cualquiera de ellos (o sobre todos) y obliga a que haya seleccionada una opción cualquiera del grupo.
- **pattern:** obliga a que el contenido del campo cumpla la **expresión regular** indicada. Por ejemplo, para un código postal sería pattern="`^[\d]{5}$`". Al final de este tema hay una pequeña introducción a las expresiones regulares en Javascript.
- **minlength / maxlength:** indica la longitud mínima/máxima del contenido del campo.
- **min / max:** indica el valor mínimo/máximo del contenido de un campo numérico.

También se producen errores de validación si el contenido de un campo no se adapta al *type* indicado (*email*, *number*, ...) o si el valor de un campo numérico no cumple con el *step* indicado.

Cuando el contenido de un campo es válido, dicho campo obtiene automáticamente la pseudoclase **:valid**

Si el contenido del campo no es válido tendrá la pseudoclase **:invalid**, lo que nos permite poner reglas en nuestro CSS para destacar dichos campo. Por ejemplo:

```
input:invalid {
    border: 2px dashed red;
}
```

La validación se realiza al enviar el formulario. Si encuentra un error se muestra dicho error, se detiene la validación del resto de campos y no se envía el formulario.

**Importante:** Tienes un pequeño resumen, y un [ejemplo de validación](#) de formularios desde HTML5, en el documento Anexo: "JavaScript – Anexo - Validar Formularios con HTML5.pdf"

## Validación mediante la API de validación de formularios

Se recomienda que estudies la siguiente documentación: <https://es.javascript.info/forms-controls>

Mediante JavaScript tenemos acceso a todos los campos del formulario, por lo que podemos hacer la validación como queramos, pero es una tarea pesada, repetitiva y que provoca código *spaghetti* difícil de leer y mantener en el futuro.

Para hacerla más simple podemos usar la [API de validación de formularios](#) de HTML5 que permite que sea el navegador quien se encargue de comprobar la validez de cada campo, pero las acciones (mostrar mensajes de error, no enviar el formulario, ...) las realizamos desde Javascript.

Esto nos da la ventaja de:

- Los requisitos de validación de cada campo están como atributos HTML de dicho campo por lo que son fáciles de ver.
- Nos evitamos la mayor parte del código dedicada a comprobar si el contenido del campo es válido. Nosotros mediante la API sólo preguntamos si se cumplen o no, y tomamos las medidas adecuadas.
- Aprovechamos las pseudo-clases **:valid** o **:invalid** que el navegador pone automáticamente a los campos, por lo que no tenemos que añadir clases en CSS para destacarlos.

Las principales propiedades y métodos que nos proporciona esta API son:

- **checkValidity()**: método que nos dice si el campo al que se aplica es o no válido. También se puede aplicar al formulario para saber si es válido o no.
- **validationMessage**: en caso de que un campo no sea válido, esta propiedad contiene el texto del error de validación proporcionado por el navegador. Si es válido, esta propiedad es una cadena vacía
- **validity**: es un objeto que tiene propiedades booleanas para saber qué requisito del campo es el que falla:
  - **valueMissing**: indica si no se cumple el atributo **required** (es decir, valdrá **true** si el campo tiene el atributo **required** pero no se ha introducido nada en él)
  - **typeMismatch**: indica si el contenido del campo no cumple con su atributo **type** (ej. **type="email"**)
  - **patternMismatch**: indica si no se cumple con el **pattern** indicado en su atributo.
  - **tooShort / tooLong**: indican si no se cumple el atributo **minlength** o **maxlength** respectivamente.
  - **rangeUnderflow / rangeOverflow**: indica si no se cumple el atributo **min / max**
  - **stepMismatch**: indica si no se cumple el atributo **step** del campo

- **customError:** indica al campo que se le ha puesto un error personalizado con **setCustomValidity**
- **valid:** indica si ese campo cumple con todas sus restricciones de validación y, por tanto, se considera válido.
- **setCustomValidity(mensaje):** añade un error personalizado al campo (que ahora ya NO será válido) con el mensaje pasado como parámetro. Para quitar este error se hace `setCustomValidity('')`.

### Ejemplo 1º de validación

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Formulario con Validación</title>
  <style>
    input:invalid {
      border: 2px dashed red;
    }

    input:valid {
      border: 2px solid greenyellow;
    }
  </style>
</head>
<body>
  <h2>Formulario de Validación</h2>

  <form id="myForm">
    <label for="nombre">Nombre:</label>
    <input type="text" id="nombre" name="nombre" >
    <br>

    <label for="apellidos">Apellidos:</label>
    <input type="text" id="apellidos" name="apellidos" required="true">
    <br>

    <label for="email">Correo electrónico:</label>
    <input type="email" id="email" name="email">
    <br>

    <input type="submit" value="Enviar">
  </form>

</body>
</html>
```

En este ejemplo, el input “apellidos” se ve con borde rojo porque el atributo “required” está activado.

Si el input “email” no es correcto, aparecen los mensajes predeterminados en HTML5

A continuación, vamos a hacer algunas modificaciones sobre este ejemplo:

### Ejemplo 2º de validación (con JavaScript)

Añadimos un script para personalizar el mensaje que aparece en el input “email”, en nuestro caso la información resultante es menos precisa que la predeterminada en HTML5.

```
<script>
    const email = document.getElementById("email");
    email.addEventListener("input", function (event) {
        if (email.validity.typeMismatch) {
            email.setCustomValidity("¡Introduzca dirección de correo electrónico!");
        } else {
            email.setCustomValidity("");
        }
    });
</script>
```

En este ejemplo la constante email guarda una referencia para el input de la dirección de correo electrónico, luego se le añade un detector de eventos que ejecuta el código de la función cada vez que el valor de la entrada cambia.

Dentro del código que contiene, verificamos si la propiedad validity.typeMismatch del input de la dirección de correo electrónico devuelve true, lo que significa que el valor que contiene no coincide con el patrón para una dirección de correo electrónico bien formada. Si es así, llamamos al método setCustomValidity() con un mensaje personalizado. Esto hace que la entrada no sea válida, de modo que cuando se intenta enviar el formulario, el envío falla y se muestra el mensaje de error personalizado.

Si la propiedad validity.typeMismatch devuelve false, se llama al método setCustomValidity() con una cadena vacía. Esto hace que la entrada sea válida y el formulario sea enviado.

### Ejemplo 3º de validación (con JavaScript y evento “onsubmit”)

La principal diferencia consiste en que se añade una función que se llama al dispararse el evento “onsubmit”

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Formulario con Validación 3</title>
    <style>
        .error {
            color: red;
        }
    </style>
</head>
<body>

    <h2>Formulario de Validación 3 </h2>

    <form id="myForm" onsubmit="return validarFormulario()">
        <label for="nombre">Nombre:</label>
        <input type="text" id="nombre" name="nombre">
        <span id="errorNombre" class="error"></span>
        <br>
```

```

<label for="apellidos">Apellidos:</label>
<input type="text" id="apellidos" name="apellidos" required="true">
<br>

<label for="email">Correo electrónico:</label>
<input type="email" id="email" name="email">
<span id="errorEmail" class="error"></span>
<br>

<input type="submit" value="Enviar">
</form>

<script>
    function validarFormulario() {
        var nombre = document.getElementById("nombre").value;
        var email = document.getElementById("email").value;

        // Validación del campo de nombre
        if (nombre === "") {
            document.getElementById("errorNombre").innerHTML = "Por favor, ingresa tu nombre.";
            return false;
        } else {
            document.getElementById("errorNombre").innerHTML = "";
        }

        // Validación del campo de correo electrónico
        if (email === "") {
            document.getElementById("errorEmail").innerHTML = "Por favor, ingresa tu correo electrónico.";
            return false;
        } else {
            document.getElementById("errorEmail").innerHTML = "";
        }

        // Otras validaciones pueden agregarse según sea necesario

        return true; // Envía el formulario si todas las validaciones son exitosas
    }
</script>

</body>
</html>

```

onsubmit es un atributo de evento del formulario que se dispara cuando el formulario está a punto de ser enviado (cuando el usuario hace clic en el botón de enviar).

validarFormulario() es una función de JavaScript que se llama cuando ocurre el evento onsubmit.

return validarFormulario(): Aquí, el return se utiliza para determinar si el formulario debe continuar con la acción predeterminada de enviar o si debe detenerse (no enviar) según el resultado de la función validarFormulario().

**Nota:** Cuando se utiliza return con una función en el atributo onsubmit, se está aprovechando la capacidad de la función para devolver un valor booleano (true o false). Este valor booleano tiene un papel crucial en el proceso de envío del formulario. Aquí está la lógica:

Si validarFormulario() devuelve true, el formulario se enviará.

Si validarFormulario() devuelve false, el formulario no se enviará.

Al agregar return, se está permitiendo que la función validarFormulario() influencie directamente el comportamiento del formulario. Si no se utiliza return, el evento onsubmit no tiene información sobre el resultado de la validación, y el formulario seguirá enviándose independientemente de si la validación es exitosa o no.

La función validarFormulario() realiza la validación del formulario y devuelve true si la validación es exitosa (es decir, si los campos cumplen con los criterios establecidos) y false si hay algún error de validación.

Resumiendo, está configurando el formulario para que, cuando se intente enviar (onsubmit), se ejecute la función validarFormulario(). Si esta función devuelve true, el formulario se enviará; de lo contrario, si devuelve false, el formulario no se enviará, proporcionando así una forma de validar los datos antes de enviarlos al servidor.

### Ejemplo para ver propiedades y métodos de la API

Veamos un ejemplo simple del valor de las diferentes propiedades involucradas en la validación de un campo de texto obligatorio y cuyo tamaño debe estar entre 5 y 50 caracteres. Prueba este código introduciendo en el input cadenas de texto de diferentes longitudes.

```
<!DOCTYPE html>
<html lang="es">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
    <style>
      .error {
        color: red;
      }
    </style>
  </head>
  <body>
    <form action="">
      <label>Nombre:</label>
      <input type="text" required minlength="5" maxlength="50" />
      <span id="error" class="error"></span>
      <br />
      <button id="comprueba">Comprueba</button>
      <p>checkValidity: <span id="checkValidity"></span></p>
      <p>validationMessage: <span id="validationMessage"></span></p>
      <p>validity.valueMissing: <span id="valueMissing"></span></p>
      <p>validity.tooShort: <span id="tooShort"></span></p>
      <p>validity.tooLong: <span id="tooLong"></span></p>
    </form>
    <script>
      document
        .getElementById("comprueba")
        .addEventListener("click", (event) => {
```

```

const inputName = document.getElementsByTagName("input")[0];

document.getElementById("error").innerHTML = inputName.validationMessage;
document.getElementById("checkValidity").innerHTML = inputName.checkValidity();
document.getElementById("validationMessage").innerHTML = inputName.validationMessage;
document.getElementById("valueMissing").innerHTML = inputName.validity.valueMissing;
document.getElementById("tooShort").innerHTML = inputName.validity.tooShort;
document.getElementById("tooLong").innerHTML = inputName.validity.tooLong;

});

</script>
</body>
</html>

```

### Ejemplo de validación usando “novalidate”

Para validar un formulario nosotros, pero usando esta API, debemos añadir al <FORM> el atributo **novalidate** que hace que el navegador no se encargue de mostrar los mensajes de error, ni de decidir si se envía o no el formulario (aunque sí valida los campos), sino que lo haremos nosotros.

*index.html*

```

<form novalidate>
  <label for="nombre">Por favor, introduzca su nombre (entre 5 y 50 caracteres): </span>
  <input type="text" id="nombre" name="nombre" required minlength="5" maxlength="50">
  <span class="error"></span>
  <br />
  <label for="mail">Por favor, introduzca una dirección de correo electrónico: </label>
  <input type="email" id="mail" name="mail" required minlength="8">
  <span class="error"></span>
  <br />
  <button type="submit">Enviar</button>
</form>

```

*main.js*

```

const form = document.getElementsByTagName('form')[0];

const nombre = document.getElementById('nombre');
const nombreError = document.querySelector('#nombre + span.error');
const email = document.getElementById('mail');
const emailError = document.querySelector('#mail + span.error');

form.addEventListener('submit', (event) => {
  if(!form.checkValidity()) {
    event.preventDefault();
  }
  nombreError.textContent = nombre.validationMessage;
  emailError.textContent = email.validationMessage;
});

```

*style.css*

```
.error {
```

```

    color: red;
}

input:invalid {
  border: 2px dashed red;
}

```

En el navegador se vería algo así:

Por favor, introduzca su nombre (entre 5 y 50 caracteres):

Por favor, introduzca una dirección de correo electrónico:

Y si se produce error en los inputs, al enviar se obtendrá algo como:

Por favor, introduzca su nombre (entre 5 y 50 caracteres):  Aumenta la longitud del texto a 5 caracteres como mínimo (actualmente, el texto tiene 3 caracteres).

Por favor, introduzca una dirección de correo electrónico:  Incluye un signo "@" en la dirección de correo electrónico. La dirección "sblanco" no incluye el signo "@".

Estamos usando:

- validationMessage para mostrar el posible error de cada campo, o quitar el error cuando el campo sea válido
- checkValidity() para no enviar/procesar el formulario si contiene errores

Si no nos gusta el mensaje del navegador y queremos personalizarlo, podemos hacer una función que reciba un <input> y, usando su propiedad validity, que devuelva un mensaje en función del error detectado:

```

function customErrorValidationMessage(input) {
  if (input.checkValidity()) {
    return ''
  }
  if (input.validity.valueMissing) {
    return 'Este campo es obligatorio'
  }
  if (input.validity tooShort) {
    return `Debe tener al menos ${input.minLength} caracteres`
  }
  // Y seguiremos comprobando cada atributo que hayamos usado en el HTML
  return 'Error en el campo' // por si se nos ha olvidado comprobar algo
}

```

Y ahora en vez de nombreError.textContent = nombre.validationMessage haremos nombreError.textContent = customErrorValidationMessage(nombre).

Si tenemos que validar algo que no puede hacerse mediante atributos HTML (por ejemplo si el nombre de usuario ya está en uso) deberemos hacer la validación “a mano” y, en caso de no ser válido, ponerle un error con .setCustomValidation().

Pero debemos recordar quitar el error si todo es correcto o el formulario siempre será inválido. Modificando el ejemplo quedaría:

```

const nombre = document.getElementById('nombre');
const nombreError = document.querySelector('#nombre + span.error');

if (nombreEnUso(nombre)) {
    nombre.setCustomValidation('Ese nombre de usuario ya está en uso')
} else {
    nombre.setCustomValidation('') // Se quita el error personalizado
}
form.addEventListener('submit', (event) => {
    if(!form.checkValidity()) {
        ...
    }
})

```

## Librerías Javascript para validación.

Existen múltiples librerías que facilitan enormemente el tedioso trabajo de validar un formulario. Al utilizar estas librerías se puede simplificar y mejorar la consistencia de la validación del formulario. Para ello, proporcionan funciones y métodos para definir reglas de validación fácilmente.

Ejemplos de librerías de validación:

- [jQuery Validate](#)
- [yup](#)
- [Validate.js](#)

## Expresiones regulares

Las expresiones regulares permiten buscar un patrón dado en una cadena de texto. Se usan mucho a la hora de validar formularios o para buscar y reemplazar texto.

En JavaScript pueden crearse expresiones regulares de dos formas:

1. Poniéndolas entre caracteres barra / (forma recomendada)
2. Instanciándolas desde la clase *RegExp*

```

let cadena='Hola mundo';
let expr=/mundo/;
expr.test(cadena);      // devuelve true porque en la cadena se encuentra la expresión 'mundo'

```

## Patrones

La potencia de las expresiones regulares es que podemos usar patrones para construir la expresión. Los más comunes son:

- **[..]** (corchetes): dentro se ponen varios caracteres o un rango y permiten comprobar si el carácter de esa posición de la cadena coincide con alguno de ellos. Ejemplos:
  - [abc]: cualquier carácter de los indicados ('a' o 'b' o 'c')
  - [^abc]: cualquiera excepto los indicados
  - [a-z]: cualquier minúscula (el carácter '-' indica el rango entre 'a' y 'z', incluidas)
  - [a-zA-Z]: cualquier letra
- **( | )** (pipe): debe coincidir con una de las opciones indicadas:
  - (x|y): la letra x o la y (sería equivalente a [xy])

- (http|https): cualquiera de las 2 palabras
- **Metacaracteres:**
  - . (punto): un único carácter, sea el que sea
  - \d: un dígito (\D: cualquier cosa menos dígito)
  - \s: espacio en blanco (\S: lo opuesto)
  - \w: una palabra o carácter alfanumérico (\W lo contrario)
  - \b: delimitador de palabra (espacio, ppio, fin)
  - \n: nueva línea
- **Cuantificadores:**
  - +: al menos 1 vez (ej. [0-9]+ al menos un dígito)
  - \*: 0 o más veces
  - ?: 0 o 1 vez
  - {n}: n caracteres (ej. [0-9]{5} = 5 dígitos)
  - {n,}: n o más caracteres
  - {n,m}: entre n y m caracteres
  - ^: al ppio de la cadena (ej.: ^[a-zA-Z] = empieza por letra)
  - \$: al final de la cadena (ej.: [0-9]\$ = que acabe en dígito)
- **Modificadores:**
  - /i: que no distinga entre Maysc y minsc (Ej. /html/i = buscará html, Html, HTML, ...)
  - /g: búsqueda global, busca todas las coincidencias y no sólo la primera
  - /m: busca en más de 1 línea (para cadenas con saltos de línea)

EJERCICIO: contruye una expresión regular para lo que se pide a continuación y pruébala con distintas cadenas:

- un código postal
- un NIF formado por 8 números, un guión y una letra mayúscula o minúscula
- un número de teléfono y aceptamos 2 formatos: XXX XX XX XX o XXX XXX XXX. El primer número debe ser un 6, un 7, un 8 o un 9

## Métodos

Los usaremos para saber si la cadena coincide con determinada expresión o para buscar y reemplazar texto:

- expr.test(cadena): devuelve **true** si la cadena coincide con la expresión. Con el modificador */g* hará que cada vez que se llama busque desde la posición de la última coincidencia. Ejemplo:

```
let str = "I am amazed in America";
let reg = /am/g;
console.log(reg.test(str)); // Imprime true
console.log(reg.test(str)); // Imprime true
console.log(reg.test(str)); // Imprime false, hay solo dos coincidencias

let reg2 = /am/gi;          // ahora no distinguirá mayúsculas y minúsculas
console.log(reg2.test(str)); // Imprime true
console.log(reg2.test(str)); // Imprime true
console.log(reg2.test(str)); // Imprime true. Ahora tenemos 3 coincidencias con este nuevo
patrón
```

- expr.exec(cadena): igual pero en vez de *true* o *false* devuelve un objeto con la coincidencia encontrada, su posición y la cadena completa:

```
let str = "I am amazed in America";
let reg = /am/gi;
console.log(reg.exec(str)); // Imprime ["am", index: 2, input: "I am amazed in America"]
console.log(reg.exec(str)); // Imprime ["am", index: 5, input: "I am amazed in America"]
console.log(reg.exec(str)); // Imprime ["Am", index: 15, input: "I am amazed in America"]
console.log(reg.exec(str)); // Imprime null
```

- cadena.match(expr): igual que exec pero se aplica a la cadena y se le pasa la expresión. Si ésta tiene el modificador /g devolverá un array con todas las coincidencias:

```
let str = "I am amazed in America";
let reg = /am/gi;
console.log(str.match(reg)); // Imprime ["am", "am", "Am"]
```

- cadena.search(expr): devuelve la posición donde se encuentra la coincidencia buscada, o -1 si no aparece.
- cadena.replace(expr, cadena2): devuelve una nueva cadena con las coincidencias de la cadena reemplazadas por la cadena pasada como 2º parámetro:

```
let str = "I am amazed in America";
console.log(str.replace(/am/gi, "xx")); // Imprime "I xx xxazed in xxerica"

console.log(str.replace(/am/gi, function(match) {
    return "-" + match.toUpperCase() + "-";
})); // Imprime "I -AM- -AM-azed in -AM-erica"
```

No vamos a profundizar más sobre las expresiones regulares. Es muy fácil encontrar por internet la que necesitemos en cada caso (para validar un e-mail, un NIF, un CP, ...). Podemos aprender más en:

- [w3schools](#)
- El anexo de Expresiones Regulares
- Y muchas otras páginas

También, hay páginas que nos permiten probar expresiones regulares con cualquier texto, como [regexp](#).

## Validar Formularios en HTML

### Índice

Validar Formularios en HTML .....	1
Índice .....	1
Required="true" .....	1
<form novalidate> .....	3
Se pueden tener dos botones uno para validar y otro que no: .....	3
Get y Post.....	4
Query String.....	4
Usando POST .....	4
Step= "any" en input number .....	6
Select multiple="true" .....	6
Fieldset .....	7
Form action.....	7
Reset .....	8
readonly ="true".....	8
disabled="true" .....	8
Parámetros ocultos – type="hidden" .....	9
Código completo del ejemplo: .....	9
Resultado en el navegador: .....	12
Información al enviar con datos: .....	12

### Required="true"

Usar **required="true"** si queremos que sea obligatorio ese campo al enviar.

```
index.html > ...
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <meta charset="UTF-8">
5      <meta http-equiv="X-UA-Compatible" content="IE=edge">
6      <meta name="viewport" content="width=device-width, initial-scale=1.0">
7      <title>Formularios en HTML</title>
8  </head>
9  <body>
10     <h1>Formularios en HTML</h1>
11     <form>
12         <label for="nombre">Nombre:</label><br/>
13         <input type="text" id="nombre" name="nombre"
14             placeholder="Escribe tu nombre" required="true"><br/>
15
16         <label for="apellido">Apellido:</label><br/>
17         <input type="text" id="apellido" name="apellido"
18             placeholder="Escribe tu apellido" required="true"/><br/>
19
20         <label for="password">Password:</label><br/>
21         <input type="password" id="password" name="password"
22             required="true"/><br/>
23
24         <input type="submit" value="Enviar"/>
25         <input type="submit" formnovalidate="formnovalidate"
26             value="Enviar sin validar"/>
27     </form>
28 </body>
29 </html>
```

Se verá algo así:

## Formularios en HTML

Nombre:

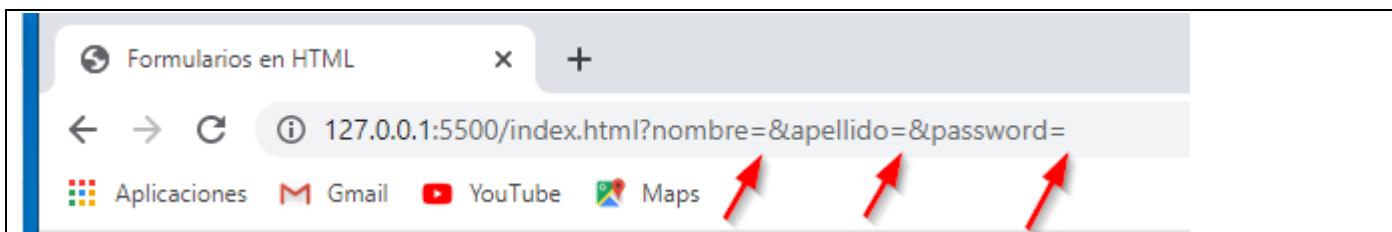
 Completa este campo

## <form novalidate>

De esta forma no se validará ningún elemento del formulario

```
<body>
  <h1>Formularios en HTML</h1>
  <form novalidate>
    <label for="nombre">Nombre:</label><br/>
    <input type="text" id="nombre" name="nombre"
      placeholder="Escribe tu nombre" required="true"/><br/>
```

Proporciona cadenas vacías al servidor.



Se pueden tener dos botones uno para validar y otro que no:

```
<body>
  <h1>Formularios en HTML</h1>
  <form> ←
    <label for="nombre">Nombre:</label><br/>
    <input type="text" id="nombre" name="nombre"
      placeholder="Escribe tu nombre" required="true"/><br/>

    <label for="apellido">Apellido:</label><br/>
    <input type="text" id="apellido" name="apellido"
      placeholder="Escribe tu apellido" required="true"/><br/>

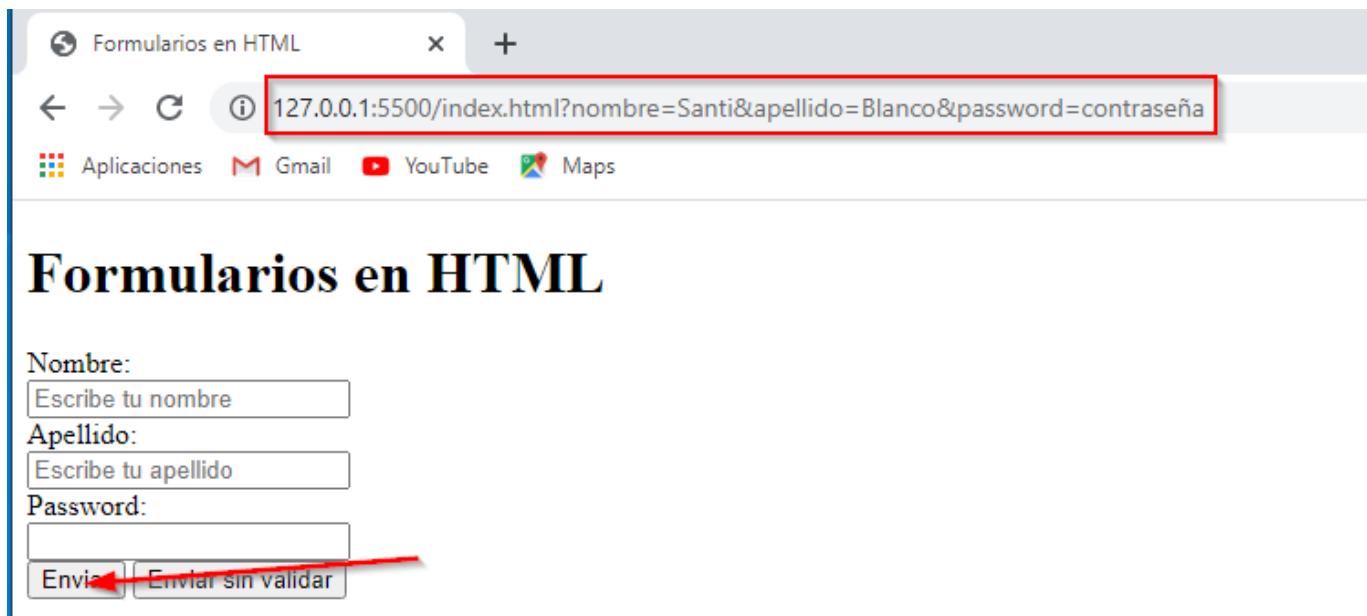
    <label for="password">Password:</label><br/>
    <input type="password" id="password" name="password"
      required="true"/><br/>

    <input type="submit" value="Enviar"/>
    <input type="submit" formnovalidate="formnovalidate"
      value="Enviar sin validar"/>
  </form>
</body>
</html>
```

# Get y Post

## Query String

Cuando estamos enviando nuestros formularios podemos observar que en la URL de nuestro navegador tenemos la URL, después del símbolo de interrogación y algunos parámetros. Es lo que se conoce como **query string**



Pero en este caso se ve la contraseña, porque estamos usando el método GET. Si queremos evitarlo hay que usar otro método para enviar información al servidor (POST).

GET: información en la URL

POST: información va dentro del cuerpo del mensaje

Para indicar uno u otro método se usa el atributo `method`, que nos ofrece 3 opciones: dialog, get y post

Si no indicamos nada, se usa el método GET, el que hemos visto en el ejemplo anterior

```
<h1>Formularios en HTML</h1>
<form method="">
  <label for="nombre">Nombre:</label>
  <input type="text" id="nombre" name="nombre" placeholder="Escribe tu nombre">
  <label for="apellido">Apellido:</label>
  <input type="text" id="apellido" name="apellido" placeholder="Escribe tu apellido">
  <label for="password">Password:</label>
  <input type="password" id="password" name="password" placeholder="Escribe tu contraseña">
  <br/>
  <input type="button" value="Enviar" /> <input type="button" value="Enviar sin validar" />
```

## Usando POST

Al darle al botón Enviar no se ve nada.

```

1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <meta charset="UTF-8">
5      <meta http-equiv="X-UA-Compatible" content="IE=edge">
6      <meta name="viewport" content="width=device-width, initial-scale=1.0">
7      <title>Formularios en HTML</title>
8  </head>
9  <body>
10     <h1>Formularios en HTML</h1>
11     <form method="post">
12         <label for="nombre">Nombre:</label><br/>
13         <input type="text" id="nombre" name="nombre"
14             placeholder="Escribe tu nombre" required="true"/><br/>
15
16         <label for="apellido">Apellido:</label><br/>
17         <input type="text" id="apellido" name="apellido"
18             placeholder="Escribe tu apellido" required="true"/><br/>
19
20         <label for="password">Password:</label><br/>
21         <input type="password" id="password" name="password"
22             required="true"/><br/>
23
24         <input type="submit" value="Enviar"/>
25         <input type="submit" formnovalidate="formnovalidate"
26             value="Enviar sin validar"/>
27     </form>
28  </body>
29  </html>

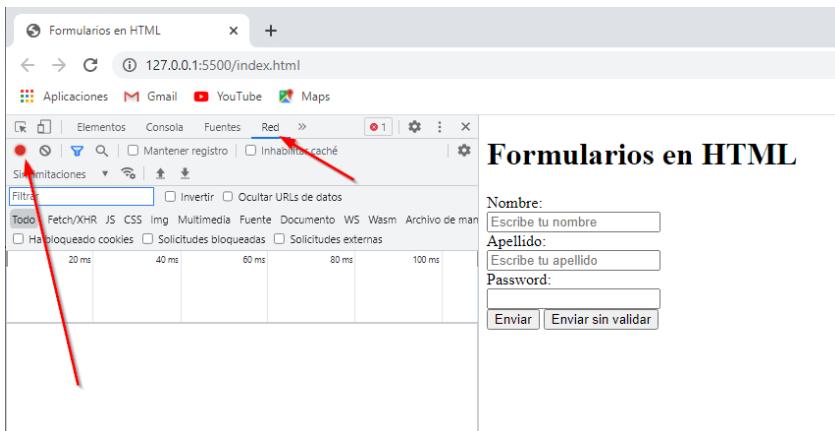
```

Esta página no tiene contenido.

Si el problema persiste, por favor inténtalo de nuevo.

HTTP ERROR 405

Si queremos verlo, inspeccionamos en el navegador la RED, y grabamos la petición con el botón de grabar **registro de red**:



The screenshot shows the Network tab in the Chrome DevTools. The URL bar at the top shows 'localhost:5500/index.html'. Below it, the 'Red' tab is highlighted. In the main list, 'index.html' is selected, and its payload is shown in the 'Payload' section, which contains form data: 'nombre: Santi', 'apellido: Blanco', and 'password: 123'.

Podemos comprobar que, aunque no ha habido respuesta por parte del servidor, se ha enviado información al servidor.

**Nota:** Para enviar información sensible se suele usar el tipo POST.

## Step= "any" en input number

Se usa para permitir decimales.

```
<input type="number" id="edad" name="edad" step="any" /><br/>
```

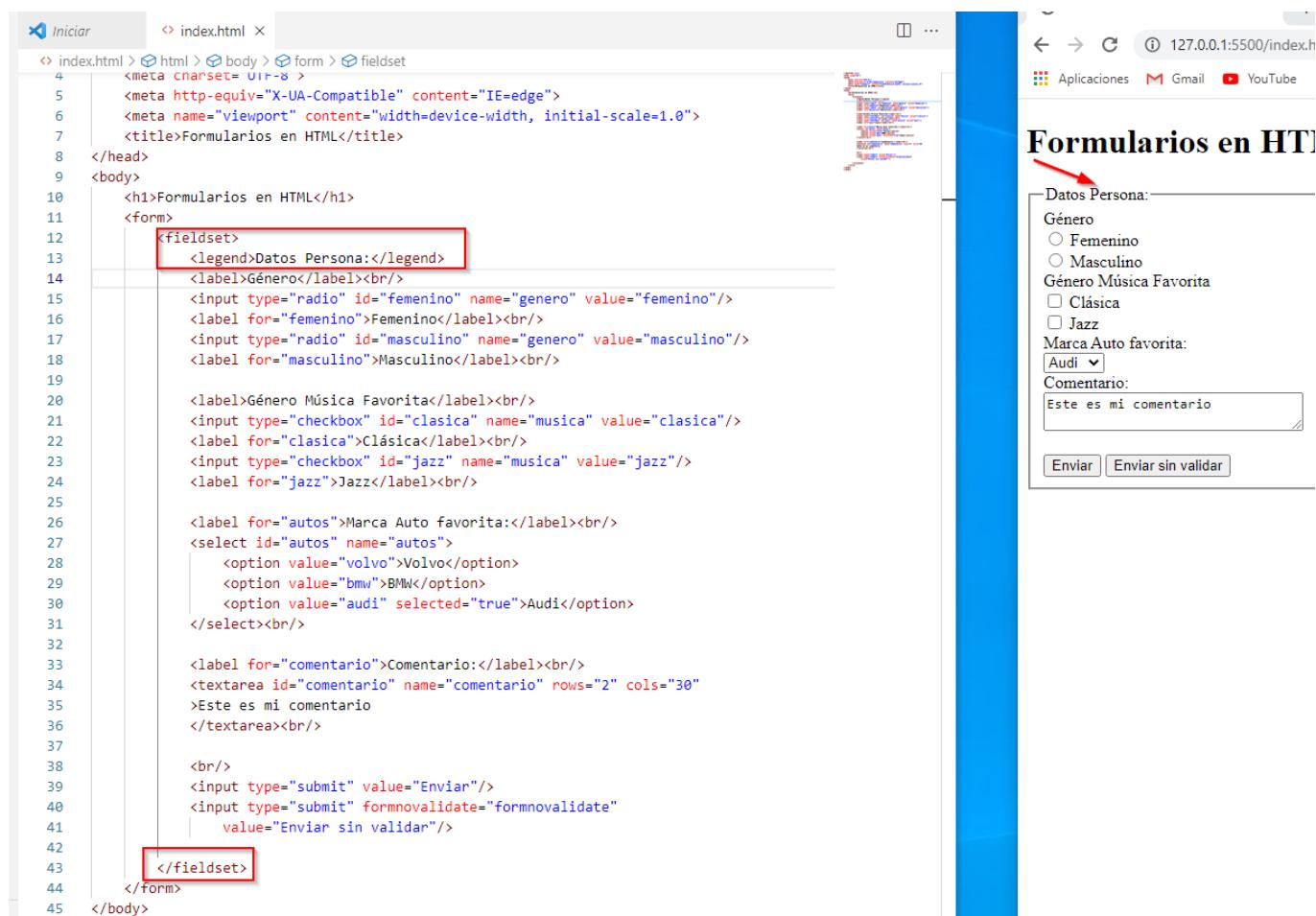
## Select multiple="true"

The screenshot shows a browser displaying a form. It includes a checkbox labeled 'Jazz' and a select element labeled 'Marca Auto favorito:' with three options: 'Volvo', 'BMW', and 'Audi'. The 'multiple="true"' attribute is highlighted in red in the code view. In the browser, an arrow points to the dropdown menu, which shows 'Volvo', 'BMW', and 'Audi' listed as selected items.

Permite seleccionar varios elementos.

Al enviar lo haría como un array.

# Fieldset



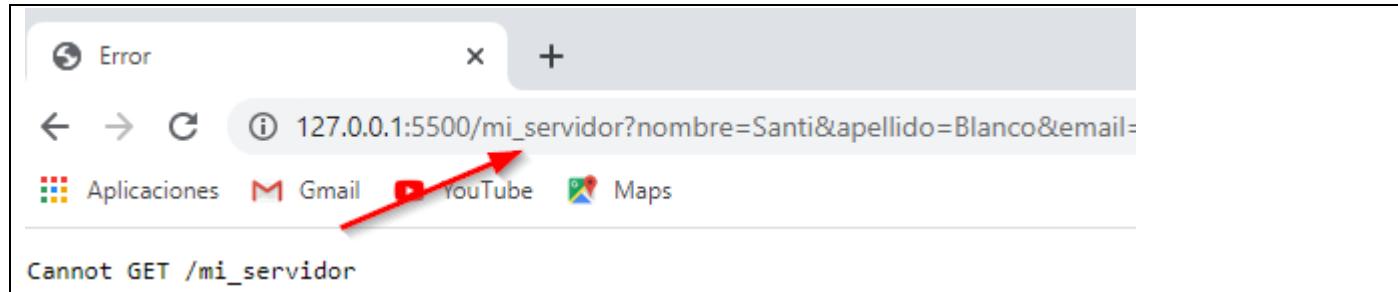
The screenshot shows a browser window with the URL `127.0.0.1:5500/index.html`. On the left, the code editor displays the HTML code for a form. A red box highlights the `<fieldset>` and `</fieldset>` tags. On the right, the browser preview shows a form titled "Formularios en HT". It contains several input fields: "Género" (radio buttons for "Femenino" and "Masculino"), "Género Música Favorita" (checkboxes for "Clásica" and "Jazz"), "Marca Auto favorita" (select dropdown with options "Volvo", "BMW", and "Audi", where "Audi" is selected), "Comentario" (text area with placeholder "Este es mi comentario"), and two submit buttons ("Enviar" and "Enviar sin validar").

Se utiliza para enmarcar distintos elementos de un formulario.

# Form action

```
<form action="mi_servidor">
    <fieldset>
        <legend>Datos Persona:</legend>
        <label for="nombre">Nombre:</label><br/>
        <input type="text" id="nombre" name="nombre"
               placeholder="Escribe tu nombre" required="true"/><br/>
```

Se utiliza para indicar cual es el componente del lado servidor o la web que va a procesar este formulario.



Sirve para los métodos POST y GET

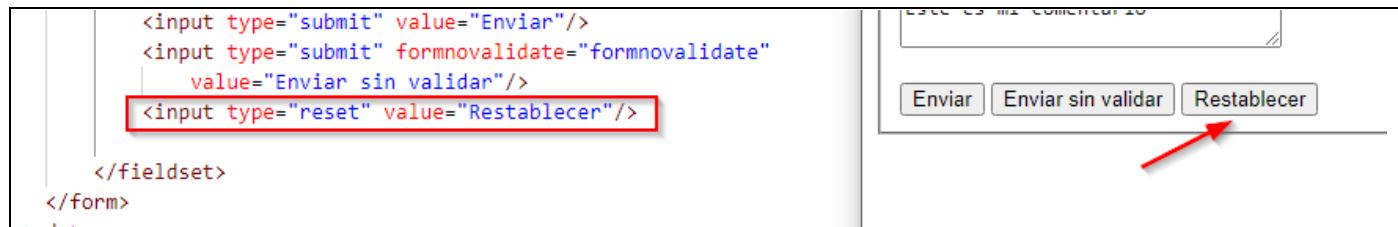
Se puede combinar con atributo **target**.

Target="\_self" o "\_blank"

Se puede combinar con atributo autocomplete="on" "off".

```
<form action="mi_servidor" target="_blank" autocomplete="off">
```

## Reset



## readonly = "true"

Cuando queremos que el elemento del formulario no lo pueda modificar el usuario, solamente es información que le aparece al usuario y que se envía desde el servidor.

El usuario puede hacer clic sobre el elemento sin poder modificar.

```

<label for="idPersona">Id Persona:</label><br/>
<input type="text" id="idPersona" name="idPersona"
       readonly="true" value="100" /><br/>

```

En el ejemplo, al enviar la información al servidor, se envía el valor "100", aunque sea de solo lectura.

## disabled="true"

Para que esté deshabilitado, no solamente que no se pueda modificar:

```
<label for="idPersona">Id Persona:</label><br/>
<input type="text" id="idPersona" name="idPersona"
       readonly="true" value="100" disabled="true"/><br/>
```

En el navegador se ve, pero al estar deshabilitado, se ve en gris difuminado y no se puede hacer clic sobre él.

Datos Persona:

Id Persona:  
100

Nombre:  
Escribe tu nombre

Apellido:  
Escribe tu apellido



Importante: Al estar deshabilitado, no se envía la información al servidor.

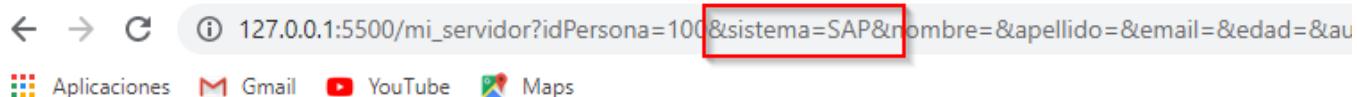
Si queremos que la información se envíe al servidor: readonly="true"

Si no queremos que la información se envíe al servidor: disabled="true"

## Parámetros ocultos – type="hidden"

```
<input type="hidden" id="sistema" name="sistema" value="SAP"/>
```

Se envía al servidor, aunque no se vea en el navegador:



## Código completo del ejemplo:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Formularios en HTML</title>
```

```
</head>
<body>
    <h1>Formularios en HTML</h1>
    <form action="mi_servidor">
        <fieldset>
            <legend>Datos Persona:</legend>

            <label for="idPersona">Id Persona:</label><br/>
            <input type="text" id="idPersona" name="idPersona"
                   readonly="true" value="100" disabled="true"/><br/>

            <input type="hidden" id="sistema" name="sistema" value="SAP"/>

            <label for="nombre">Nombre:</label><br/>
            <input type="text" id="nombre" name="nombre"
                   placeholder="Escribe tu nombre" required="true"/><br/>

            <label for="apellido">Apellido:</label><br/>
            <input type="text" id="apellido" name="apellido"
                   placeholder="Escribe tu apellido" required="true"/><br/>

            <label for="email">Email:</label><br/>
            <input type="email" id="email" name="email"/><br/>

            <label for="edad">Edad:</label><br/>
            <input type="number" id="edad" name="edad"/><br/>

            <label>Género</label><br/>
            <input type="radio" id="femenino" name="genero" value="femenino"/>
            <label for="femenino">Femenino</label><br/>
            <input type="radio" id="masculino" name="genero" value="masculino"/>
            <label for="masculino">Masculino</label><br/>

            <label>Género Música Favorita</label><br/>
            <input type="checkbox" id="clasica" name="musica" value="clasica"/>
            <label for="clasica">Clásica</label><br/>
            <input type="checkbox" id="jazz" name="musica" value="jazz"/>
            <label for="jazz">Jazz</label><br/>
```

```
<label for="autos">Marca Auto favorita:</label><br/>
<select id="autos" name="autos">
    <option value="volvo">Volvo</option>
    <option value="bmw">BMW</option>
    <option value="audi" selected="true">Audi</option>
</select><br/>

<label for="comentario">Comentario:</label><br/>
<textarea id="comentario" name="comentario" rows="2" cols="30">
Este es mi comentario
</textarea><br/>

<br/>
<input type="submit" value="Enviar"/>
<input type="submit" formnovalidate="formnovalidate"
       value="Enviar sin validar"/>
<input type="reset" value="Restablecer"/>

</fieldset>
</form>
</body>
</html>
```

## Resultado en el navegador:

The screenshot shows a web browser window with the title "Formularios en HTML". The address bar displays the URL "127.0.0.1:5500/index.html". Below the address bar are links for "Aplicaciones", "Gmail", "YouTube", and "Maps". The main content area contains the following form fields:

- Datos Persona:**
- Id Persona:**
- Nombre:**
- Apellido:**
- Email:**
- Edad:**
- Género**

  - Femenino
  - Masculino

- Género Música Favorita**

  - Clásica
  - Jazz

- Marca Auto favorita:**
- Comentario:**

At the bottom of the form are three buttons: "Enviar", "Enviar sin validar", and "Restablecer".

## Información al enviar con datos:

The screenshot shows a browser window with the title "Error". The address bar displays the URL "127.0.0.1:5500/mi\_servidor?sistema=SAP&nombre=Santi&apellido=Blanco&email=sblanco%40educa.jcyl.es&edad=27&genero=masculino&musica=clasica&musica=jazz&autos=volvo&comentario=Este+es+mi+comentario++++%0D%0A+++++++++++".

The URL is also displayed below the browser window:

```
http://127.0.0.1:5500/mi\_servidor?sistema=SAP&nombre=Santi&apellido=Blanco&email=sblanco%40educa.jcyl.es&edad=27&genero=masculino&musica=clasica&musica=jazz&autos=volvo&comentario=Este+es+mi+comentario++++%0D%0A+++++++++++
```



## DWEC – Javascript Web Cliente.

JavaScript – Formato JSON .....	1
Introducción.....	1
Características del formato .....	1
Ejemplos.....	1
Estructura de los datos .....	3
Métodos del objeto JSON .....	3

# JavaScript – Formato JSON

## Introducción

El formato JSON (acrónimo de JavaScript Object Notation, 'notación de objeto de JavaScript') es una forma de convertir objetos Javascript en una cadena de texto para poderlos enviar.

## Características del formato

- La estructura de datos JSON se compone de un conjunto de objetos o arrays que contendrán números, cadenas booleanos y nulos.
- Un **objeto** JSON comienza y termina con llaves, y contiene una colección desordenada de pares **nombre**-**valor**.
- Cada **nombre** y **valor** están separados por dos puntos, y los pares están separados por comas.
- La coma final está prohibida.
- El **nombre** es una cadena entre comillas dobles. Los caracteres de comillas no deben ser inclinadas o "inteligentes".
- En los **números**, los ceros a la izquierda están prohibidos; un punto decimal debe estar seguido al menos por un dígito.
- En las cadenas deben estar entre comillas dobles. No se permiten todos los caracteres de escape; sí se permiten los caracteres de separador de línea Unicode (U+2028) y el separador de párrafo (U+2029).
- Un array JSON comienza y termina con corchetes y contiene una colección ordenada de valores separados por comas. Un valor puede ser una cadena entre comillas dobles, un número, un booleano true o false, nulo, un objeto JSON o un array.
- Los objetos y los arrays JSON se pueden anidar, lo que posibilita una estructura jerárquica de datos.

## Ejemplos

En el siguiente ejemplo, se muestra una estructura de datos JSON con dos objetos válidos.

```
{  
  "id": 1006410,  
  "title": "Amazon Redshift Database Developer Guide"  
}
```

```
{
  "id": 100540,
  "name": "Amazon Simple Storage Service User Guide"
}
```

En el siguiente se muestran los mismos datos como dos arrays JSON:

```
[
  1006410,
  "Amazon Redshift Database Developer Guide"
]
[
  100540,
  "Amazon Simple Storage Service User Guide"
]
```

### Conversiones de objetos

El objeto alumno:

```
let alumno = {
  id: 5,
  nombre: 'Ana',
  apellidos: 'Zubiri Peláez'
}
```

se transformaría en la cadena de texto:

```
{ "id": 5, "nombre": "Ana", "apellidos": "Zubiri Peláez" }
```

y el array:

```
let alumnos = [
  {
    id: 5,
    nombre: "Ana",
    apellidos: "Zubiri Peláez"
  },
  {
    id: 7,
    nombre: "Carlos",
    apellidos: "Pérez Ortíz"
  },
]
```

Se transformaría en la cadena:

```
[{ "id": 5, "nombre": "Ana", "apellidos": "Zubiri Peláez" }, { "id": 7, "nombre": "Carlos", "apellidos": "Pérez Ortíz" }]
```

Nótese que tanto las claves como los valores van entrecomillados (con comillas dobles). No sirven comillas simples.

## Estructura de los datos

Los mismos datos pueden tener distinta estructura.

Archivo colores1.json	Archivo colores2.json	Archivo colores3.json
<pre>{   "arrayColores": [     {       "nombreColor": "rojo",       "valorHexadec": "#f00"     },     {       "nombreColor": "verde",       "valorHexadec": "#0f0"     },     {       "nombreColor": "azul",       "valorHexadec": "#00f"     },     {       "nombreColor": "cyan",       "valorHexadec": "#0ff"     },     {       "nombreColor": "magenta",       "valorHexadec": "#f0f"     },     {       "nombreColor": "amarillo",       "valorHexadec": "#ff0"     },     {       "nombreColor": "negro",       "valorHexadec": "#000"     }   ] }</pre>	<pre>{   "arrayColores": [     {       "rojo": "#f00",       "verde": "#0f0",       "azul": "#00f",       "cyan": "#0ff",       "magenta": "#f0f",       "amarillo": "#ff0",       "negro": "#000"     }   ] }</pre>	<pre>{   "rojo": "#f00",   "verde": "#0f0",   "azul": "#00f",   "cyan": "#0ff",   "magenta": "#f0f",   "amarillo": "#ff0",   "negro": "#000" }</pre>

Los ejemplos anteriores representan lo que podrían ser archivos JSON conteniendo datos en formato JSON.

Se trata de 3 archivos que contienen aproximadamente la misma información. Sin embargo, hay algunas diferencias:

- En el archivo **colores1.json** existe un único objeto de datos donde el nombre es *arrayColores* y su valor es un array de objetos JSON. Cada objeto del array está formado por los pares (*nombreColor* y su *valor*), y (*valorHexadec* y su *valor*). En este ejemplo en concreto el array consta de 7 elementos con información correspondiente a 7 colores.
- En el archivo **colores2.json** existe un único objeto de datos donde el nombre es *arrayColores*, cuyo valor es un array que contiene un único objeto JSON formado por siete pares (*nombre* – *valor*) que representa información sobre siete colores.
- En el archivo **colores3.json** existe un único objeto de datos que está formado por siete pares (*nombre* – *valor*) que representa información sobre siete colores.

Siendo las 3 formas válidas, se deberá utilizar aquella que se indique en las instrucciones o, de no existir pautas precisas, utilizar aquel diseño que favorezca el desarrollo y mantenimiento de la aplicación.

## Métodos del objeto JSON

Para convertir objetos en cadenas de texto JSON y viceversa, Javascript proporciona 2 métodos:

- **JSON.stringify(*objeto*)**: recibe un objeto JS y devuelve la cadena de texto correspondiente.

```
// convierte de formato JSON a objeto
const cadenaAlumnos = JSON.stringify(alumnos)
```

- **JSON.parse(*cadena*)**: realiza el proceso inverso, convirtiendo una cadena de texto en un objeto.

```
// convierte un objeto a formato JSON
```

```
const alumnos = JSON.parse(cadenaAlumnos)
```



## DWEC - Javascript Web Cliente.

JavaScript – document.cookie .....	1
Introducción:.....	1
Persistencia de la información .....	1
Concepto de cookie.....	2
Tiempo de vida de las cookies.....	3
Contenidos de las cookies.....	4
Cookies con Javascript: document.cookie .....	4
Crear una cookie .....	4
Modificar una cookie .....	6
Eliminar cookies .....	6
Recuperar el contenido de cookies .....	7
Saber si las cookies están habilitadas o no.....	7
Ejercicio HacerAlgoUnaSolaVez .....	8

# JavaScript – cookies y document.cookie

## Introducción:

Una cookie es un archivo que una página web guarda en el ordenador del cliente, y proporciona un sistema para que desde el servidor se pueda identificar al cliente sin necesidad de registrarse en el servidor.

En esta web hay buena información sobre los tipos de cookies: <https://www.xataka.com/basics/que-cookies-que-tipos-hay-que-pasa-desactivas>

## Persistencia de la información

Durante la navegación por una aplicación web, el usuario podrá visitar numerosos documentos HTML que se corresponderán con diferentes urls. Como las variables en JavaScript tienen una vida limitada al propio documento HTML en el que van insertas o enlazadas y su información desaparece cuando se carga un nuevo documento HTML, a menudo se necesita un sistema para disponer del valor establecido en algunas variables desde otro documento.

Cargar un archivo común donde definamos unas variables nos puede servir para disponer de dichas variables con su valor inicial en diferentes urls, pero si el valor de esa variable se modifica y continúa la navegación, no dispondremos de información sobre dicha modificación en el resto de urls, ya que “se perderán”.

Lo que vamos buscando es la persistencia de la información, es decir, que no desaparezca la información cuando cambiamos de url.

Hay diferentes maneras de abordar este problema que trataremos en éste y otros capítulos:

- Desde el lado cliente:
  - Utilizando cookies.

- Utilizando almacenamiento en el navegador (localStorage y sessionStorage).
- Desde el lado servidor:
  - Transmitiendo y almacenando información en una base de datos para recuperarla posteriormente.

## Concepto de cookie

Las cookies fueron creadas por trabajadores de la empresa Netscape como forma de dar una respuesta sencilla a la necesidad de almacenar información relacionada con la identificación de usuarios y acciones que un usuario desarrolla durante la navegación. Por ejemplo, se planteaba la siguiente cuestión: si un usuario accede a una tienda web y queremos que pueda ir agregando productos a un carrito de compras, ¿cómo saber qué productos ha almacenado cuando cambia de url? Sin el uso de alguna forma de dotar de persistencia a la información, la información se perdía. Y cierta información sería muy costoso manejarla con herramientas como bases de datos.

La forma de solucionarlo fue inventar las cookies. Las cookies son información que se almacena en el navegador de forma persistente, es decir, que no desaparece cuando el usuario navega a través de diferentes urls. Además, las cookies son enviadas al servidor cuando el usuario hace una petición, de modo que, el servidor puede conocer esa información y actuar en consecuencia.

Las cookies pueden ser creadas de diferentes maneras, por ejemplo:

- a) **Ser creadas por el servidor**, y enviarlas al navegador para que las almacene. Supongamos que entramos en una página web de compras por internet. En ese momento somos usuarios anónimos, pero una vez introduzcamos nuestro nombre de usuario y password (por ejemplo, supongamos que somos el usuario Albert Einstein), el servidor envía una cookie al navegador que podría ser:

`session_id_ = 6n4465736gf9863b52e641757fa0b7db`, donde `session_id` es el nombre la cookie y la cadena de letras y números el valor que tiene la cookie, lo que permitirá saber al servidor que quien hace una petición es la misma persona (el mismo computador cliente) que había hecho una petición anteriormente. La comprobación de que sea el mismo computador cliente (realmente se refiere al mismo navegador cliente) se basa en comparar el valor de esa cadena larga: si coincide, es el mismo usuario, si es diferente, es otro usuario.

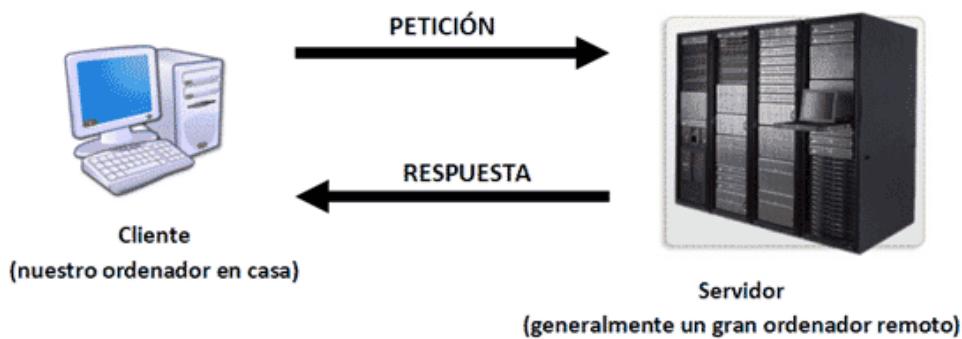
Un usuario no puede conocer el valor de la cookie que tiene otro debido a que existen prácticamente infinitas posibilidades de combinación de letras y números. Si se usan cadenas arbitrariamente largas, es prácticamente imposible averiguar por casualidad o mediante pruebas el contenido de una cookie.

Ahora bien, cada vez que cambiemos de url esta cookie es enviada al servidor. El servidor lee la cookie y comprueba: ¡esta cookie pertenece al usuario Albert Einstein!

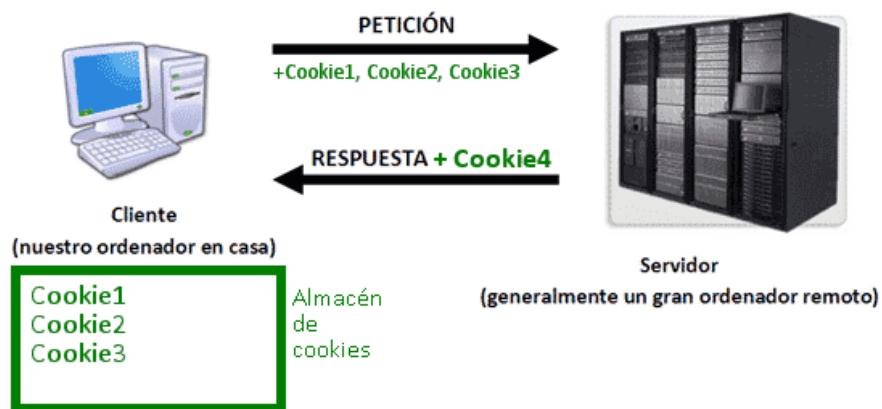
Aunque el usuario haya cambiado de url, la cookie informa en cada momento que él es Albert Einstein, con lo cual no hace falta estar introduciendo en cada url que se visite el nombre de usuario y password.

- b) **Ser creadas mediante JavaScript** en el navegador, almacenarse en el navegador y enviarse posteriormente al servidor en cada comunicación que tenga lugar.

Simplificadamente podríamos decir que navegar sin cookies sería algo similar a lo que se muestra en este esquema, donde cada petición al servidor equivale a una “petición nueva”:



Mientras que navegar con cookies sería algo similar a lo que se muestra en este esquema, donde el navegador está almacenando información en un almacén de cookies y con cada petición que hace esas cookies son transmitidas (y además pueden irse añadiendo más cookies a las ya existentes previamente):



**Resumiendo:** las cookies son datos que se almacenan en el navegador del usuario.

Las cookies pueden presentar algunos problemas de seguridad, pero en general se considera que si se utilizan de forma adecuada son un mecanismo que resulta práctico para realizar muchas de las tareas que normalmente se requieren en la navegación web.

## Tiempo de vida de las cookies

Las cookies son datos temporales, es decir, su intención no es almacenarse “para siempre”, sino almacenarse por un tiempo para facilitar la navegación. Una cookie puede tener asociada una fecha de borrado o expiración, en cuyo caso permanecerá en el navegador del usuario hasta que llegue dicha fecha (a no ser que el usuario decida hacer un borrado de cookies). En este caso, puede ocurrir que el usuario cierre el navegador y lo abra al cabo de unas horas o días, y la información en forma de cookies siga estando ahí. Las cookies con fecha de borrado se suelen llamar cookies persistentes, porque no se destruyen excepto cuando llega la fecha de expiración.

Otras cookies no tienen fecha de borrado o expiración, o si la tienen es muy corta (pongamos que una hora de duración). Si la cookie no tiene fecha de borrado, se destruye cuando se cierra el navegador.

Hay que tener en cuenta que los navegadores pueden almacenar información de otras maneras además de como cookies (por ejemplo, como opciones de configuración, perfiles de usuario, contraseñas, etc.).

Además, las cookies (al igual que JavaScript) pueden desactivarse en los navegadores. La mayoría de los usuarios navegan con cookies activadas (al igual que con JavaScript activado), pero teóricamente un usuario puede deshabilitarlas.

## Contenidos de las cookies

Las cookies podemos verlas como pequeños ficheros de texto que se almacenan en el navegador, cuyo contenido es el siguiente:

1. Un par nombre – valor que define la cookie. Por ejemplo, el nombre puede ser **user\_name** y el valor **pelaez**.
2. Una fecha de caducidad (en algunos casos, estará indefinida, con lo cual la cookie será borrada cuando se cierre el navegador). La fecha se expresa en tiempo UTC, o como un número de segundos desde el momento actual.
3. El dominio y ruta del servidor donde la cookie fue definida, lo que permite que si existieran dos cookies con el mismo nombre se pudiera saber qué cookie corresponde a cada url que se visita. No está permitido falsear dominios, es decir, si el dominio desde el que se establece una cookie es daw2cliente.com, no se podría poner como información asociada a la cookie que el dominio es microsoft.com, sino que la cookie únicamente puede ir asociada a daw2cliente.com. Esto permite que, si se está navegando por un sitio, no haya necesidad de enviar todas las cookies almacenadas en el navegador al servidor de ese sitio, sino únicamente las cookies relacionadas con ese sitio.

La ruta permite especificar un directorio específico al cual se debe considerar asociada la cookie. De este modo, el navegador no tendría que enviar la cookie a todas las páginas de un dominio, sino solo a las páginas concretas cuya ruta esté definida. Normalmente la ruta definida es simplemente <> / >> lo que significa que la cookie es válida en todo el dominio.

## Cookies con JavaScript: document.cookie

Las cookies podemos verlas como pequeños ficheros de texto que se almacenan en el navegador pero que desde el punto de vista de Javascript es una cadena de texto que contiene parejas *clave=valor* separadas por ; (punto y coma).

```
<nombre>=<valor>; expires=<fecha>; max-age=<segundos>; path=<ruta>; domain=<dominio>; secure;
```

Para obtener todas las cookies accesibles desde una localización se utiliza la propiedad:

```
let todasLasCookies = document.cookie;
```

### Crear una cookie

Para crear una cookie con JavaScript usaremos la siguiente sintaxis:

```
document.cookie=nuevaCookie;
```

donde nuevaCookie es una cadena de texto con pares clave-valor separadas por punto y coma.

*Los atributos son:*

<nombre>=<valor>

Requerido. <nombre> es el nombre (key) que identifica la cookie y <valor> es su valor. A diferencia de las cookies en PHP, en JavaScript se puede crear una cookie con un valor vacío (<nombre>=).

**expires=<fecha> y max-age=<segundos>**

Opcional. Ambos parámetros especifican el tiempo de validez de la cookie. expires establece una fecha (ha de estar en formato UTC) mientras que max-age establece una duración máxima en segundos. max-

age toma preferencia sobre expires. Si no se especifica ninguno de los dos se creará una **session cookie**. Si es max-age=0 o expires=fechaPasada la cookie se elimina.

#### path=<ruta>

Opcional. Establece la ruta para la cual la cookie es válida. Si no se especifica ningún valor, la cookie será válida para la ruta la página actual.

#### domain=<dominio>

Opcional. Dentro del dominio actual, se puede indicar el subdominio para el que la cookie es válida. El valor predeterminado es el subdominio actual. Establecer domain=.miweb.com para una cookie que sea válida para cualquier subdominio (nota el punto delante del nombre del dominio). Por motivos de seguridad, los navegadores no permiten crear cookies para dominios diferentes al que crea la cookie (*same-origin policy*).

#### secure

Opcional. Atributo sin valor. Si está presente la cookie sólo es válida para conexiones encriptadas (por ejemplo, mediante protocolo HTTPS).

#### Ejemplos de creación de cookie:

```
document.cookie= "nombreCookie=valorCookie; expires=fechaDeExpiración; path=rutaParaLaCookie";
```

Se pueden añadir a la cadena de texto otros parámetros clave-valor.

También se puede dejar sin especificar **expires** (la cookie se borrará al cerrar el navegador) y **path** (la cookie quedará asociada al dominio), con lo que la definición quedaría:

```
document.cookie = 'nombreCookie=valorCookie; ';
```

Las cookies se envían en las cabeceras HTTP y, por tanto, deben estar correctamente codificadas. Conviene utilizar encodeURIComponent() para evitar sorpresas. Ejemplo:

```
let testvalue = "Hola mundo!";
document.cookie = "nombreCookie=" + encodeURIComponent( testvalue );
```

Si se va a utilizar el atributo **expires**, ha de ser con una fecha en formato UTC. Puede ser de ayuda el método Date.toUTCString(). Por ejemplo, una cookie con caducidad para el 1 de Febrero del año 2068 a las 11:20:

```
let expiresdate = new Date(2068, 1, 02, 11, 20);
let cookievalue = "Hola Mundo!";
document.cookie = "testcookie=" + encodeURIComponent( cookievalue ) + "; expires=" +
expiresdate.toUTCString();
```

En lugar de **expires** se puede usar **max-age**. En este caso, en lugar de especificar una fecha concreta se especifica el número de segundos desde la creación de la cookie hasta su caducidad. Por ejemplo:

```
document.cookie = "color=blue; max-age=" + 60*60*24*30 + "; path=/; domain=daw2cliente.com; secure"
```

El ejemplo anterior serviría para indicar que el nombre de la cookie es *color*, su valor *blue*, su fecha de expiración 30 días (expresado en segundos resulta 30 días \* 24 horas/día \* 60 minutos/hora \* 60 segundos/minuto").

Si no se especifica max-age ni expires, la cookie expirará al terminar la sesión actual. Si se especifican ambos atributos, tiene prioridad max-age.

## La propiedad document.cookie

Cuando se van creando cookies, éstas se van añadiendo a document.cookie (es decir, document.cookie no funciona como una propiedad que se vaya sobrescribiendo, sino que cada definición de document.cookie añade una cookie a la colección de cookies del documento). Este comportamiento se debe a que document.cookie no es un dato con un valor, sino una propiedad de acceso con métodos set y get nativos. Cada vez que se le asigna una nueva cookie, no se sobrescriben las cookies anteriores, sino que la nueva se añade a la colección de cookies del documento.

## Modificar una cookie

Una cookie se puede redefinir: para ello simplemente hemos de usar document.cookie indicando el mismo nombre de cookie que existiera previamente. Los datos de esa cookie serán sobrescritos, quedando reemplazados los anteriormente existentes por los nuevos.

Es importante tener en cuenta que, si una cookie se crea para un dominio o para un path determinado y se quiere modificar, **el dominio y el path han de coincidir**. De lo contrario se crearán dos cookies diferentes válidas para cada path y dominio. Por ejemplo, imaginemos que estamos en «miweb.com/blog» (el valor predeterminado del path es en este caso /blog):

```
// Supongamos que estamos en "miweb.com/blog"
// y creamos las siguientes cookies

// Creamos la cookie para el path "/"
document.cookie = "nombre=Miguel; path=/";

// Con la siguiente linea se crea una nueva cookie para el path "/blog" (valor por defecto)
// pero no se modifica la cookie "nombre" anterior porque era para un path diferente
document.cookie = "nombre=Juan";

// Con la siguiente línea SÍ se modifica la cookie "nombre" del path "/" correctamente
document.cookie = "nombre=Juan; path=/";
```

## Eliminar cookies

Una cookie se puede eliminar: para ello hemos de usar document.cookie indicando el nombre de cookie que queremos eliminar y establecer una fecha de expiración ya pasada (por ejemplo del día de ayer, o simplemente indicar **expires=Thu, 01 Jan 1970 00:00:00 UTC**). El navegador al comprobar que la fecha de caducidad de la cookie ha pasado, la eliminará.

Por ejemplo, creamos la cookie con el identificador nombre y valor Miguel igual que antes:

```
document.cookie = "nombre=Miguel";
```

Tenemos dos formas de eliminarla:

```
document.cookie = "nombre=; expires=Thu, 01 Jan 1970 00:00:00 UTC";

// O con max-age
document.cookie = "nombre=; max-age=0";
```

## Recuperar el contenido de cookies

Para recuperar el contenido de cookies hemos de trabajar con `document.cookie` como si fuera una cadena de texto

Para recuperar el valor de la cookie, hemos de buscar dentro de esa cadena de texto (string). Para ello usaremos las herramientas que nos proporciona JavaScript.

```
let todasLasCookies = document.cookie;
```

En la variable `todasLasCookies` las cookies se organizan de la siguiente manera:

```
nombreCookieA=valorCookieA; nombreCookieB=valorCookieB; ... ; nombreCookieN = valorCookieN;
```

A tener en cuenta:

- El string sólo contiene pares de nombre de la cookie y su valor. No se puede acceder a otros parámetros a través de `document.cookie`.
- Sólo se obtienen las cookies válidas para el documento actual. Esto implica que cookies para otros paths, dominios o cookies caducadas no se pueden leer. Aunque en una página puedan crearse cookies para otros subdominios y paths, sólo se pueden leer las que sean válidas para el subdominio y path actual.

Por ejemplo, imagina que estamos en el subdominio `noticias.miweb.com`. Aquí podemos crear una cookie para el subdominio `tienda.miweb.com`, pero **esta cookie no es válida para el documento en el que estamos** (`noticias.miweb.com`), por lo que no podemos leer su valor desde aquí, aunque sí hemos podido crearla:

```
// Suponiendo que estamos en noticias.miweb.com

// Se crean dos cookies para dos subdominios diferentes
document.cookie = "cookienoticias=valorcn; domain=noticias.miweb.com";
document.cookie = "cookietienda=valorct; domain=tienda.miweb.com";

let lasCookies = document.cookie;
alert( lasCookies );
// Obtenemos cookienoticias=valorcn
// No podemos acceder a la cookie cookietienda
// porque es válida solo para tienda.miweb.com y estamos en noticias.miweb.com
```

Para leer el contenido de cookies individuales se hace manipulando el string con todas las cookies y dividirlo por cada `;` para separar cada par `nombrecookie=valor`. Luego se divide cada uno de estos pares por `=` para separar el nombre de la cookie y su valor. Se puede conseguir utilizando varios métodos.

Ejercicio: Realiza y prueba una función que obtenga el valor de una cookie pasando su nombre como parámetro.

Ejercicio: utilizando expresiones regulares, realiza y prueba una función que obtenga el valor de una cookie pasando su nombre como parámetro.

## Saber si las cookies están habilitadas o no.

Un usuario puede eliminar las cookies de su navegador. También puede desactivar las cookies. Por ello, puede ser interesante saber si están activadas, se sabe leyendo el objeto `navigator` y la propiedad booleana `cookieEnabled`.

Ejercicio: Realiza una página web que muestre un mensaje en el documento indicando si las cookies están o no activadas. Prueba la función activando y desactivando las cookies de tu navegador.

## Ejercicio: HacerAlgoUnaSolaVez

```
/*
Para usar el siguiente código: reemplaza todas las veces la
palabra hacerAlgoUnaSolaVez (el nombre de la cookie) con un nombre personalizado.
*/
function hazUnaVez() {
    if (document.cookie.replace(/(?:^(?:\s*; )*)hacerAlgoUnaSolaVez\s*\=\s*([^;]*).*$/, "$1") !== "true") {
        alert("Hacer algo aquí!");
        document.cookie = "hacerAlgoUnaSolaVez=true; expires=Fri, 31 Dec 9999 23:59:59 GMT";
    }
}

<button onclick="hazUnaVez()">Solo hacer algo una vez</button>
```



## DWEC – Javascript Web Cliente.

JavaScript – Almacenamiento web HTML.....	1
Introducción.....	1
Objetos de almacenamiento web HTML.....	1
Métodos y propiedades disponibles: .....	2
El objeto de almacenamiento local (localStorage).....	2
Almacenar y recuperar información .....	2
Eliminar información.....	3
Eliminar todos los datos .....	3
Ejemplo: .....	3
El objeto de almacenamiento (sessionStorage).....	3
Ejemplo: .....	4
Persistencia de la información almacenada.....	4
El evento storage .....	4
Diferencias entre cookies y storage .....	4
Ejemplo: Almacenar un array de objetos con JSON y recuperarlo.....	5
Usando Object.create() .....	8
Obteniendo el array de objetos de la clase Persona: .....	8

# JavaScript – Almacenamiento web HTML

## Introducción

Con el almacenamiento web HTML las aplicaciones web pueden almacenar datos localmente dentro del navegador del usuario.

Antes de HTML5, los datos de la aplicación tenían que almacenarse en cookies.

El almacenamiento web es más seguro y se pueden almacenar grandes cantidades de datos localmente, sin afectar el rendimiento del sitio web.

A diferencia de las cookies, el límite de almacenamiento es mucho mayor (al menos 5 MB), y la información nunca se transfiere al servidor.

El almacenamiento web queda definido por su origen (por su dominio y el protocolo utilizado). Si el usuario cambia de página, el almacén de datos es distinto.

## Objetos de almacenamiento web HTML

El almacenamiento web HTML proporciona dos objetos para almacenar datos en el cliente:

- `window.localStorage`: almacena datos sin fecha de caducidad.

- `window.sessionStorage`: almacena datos para una sesión (los datos se pierden cuando se cierra la pestaña del navegador).

Antes de usar el almacenamiento web es aconsejable comprobar la compatibilidad del navegador con `localStorage` y `sessionStorage`:

```
if (typeof(Storage) !== "undefined") {
    // Code for LocalStorage/sessionStorage.
} else {
    // Sorry! No Web Storage support..
}
```

## Métodos y propiedades disponibles:

En la siguiente tabla se describen los métodos y propiedades para el objeto `window.sessionStorage`. El objeto `window.localStorage` utiliza los mismos.

Método o propiedad de <code>sessionStorage</code>	Descripción
<code>sessionStorage.setItem('clave', 'valor');</code>	Guarda la información <b>valor</b> a la que se podrá acceder invocando a <code>clave</code> . Por ejemplo, <code>clave</code> puede ser nombre y <code>valor</code> puede ser Carlos.
<code>sessionStorage.getItem('clave')</code>	Recupera el <i>value</i> de la clave especificada. Por ejemplo, si <code>clave</code> es nombre puede recuperar “Carlos”.
<code>sessionStorage[clave]=valor</code>	Igual que <code>setItem</code>
<code>sessionStorage.length</code>	Devuelve el número de items guardados por el objeto <code>sessionStorage</code> actual.
<code>sessionStorage.key(i)</code>	Cada item se almacena con un índice que comienza por cero y se incrementa unitariamente por cada item añadido. Con esta sintaxis rescatamos la clave correspondiente al item con índice <code>i</code> .
<code>sessionStorage.removeItem(clave)</code>	Elimina un item almacenado en <code>sessionStorage</code>
<code>sessionStorage.clear()</code>	Elimina todos los items almacenados en <code>sessionStorage</code> , quedando vacío el espacio de almacenamiento.

## El objeto de almacenamiento local (`localStorage`)

El objeto **localStorage** almacena los datos sin fecha de caducidad. Los datos no se eliminarán cuando se cierre el navegador y estarán disponibles al día, semana o año siguiente.

### Almacenar y recuperar información

Se utilizan los métodos `localStorage.setItem()` y `localStorage.getItem()`

Los pares de nombre/valor siempre se almacenan como cadenas. Hay que convertirlos a otro formato cuando sea necesario.

Ejemplo para:

- Crear un par de nombre/valor de almacenamiento local con nombre="apellido" y valor="Smith"
- Recuperar el valor de "apellido" e insértelo en el elemento con id="resultado"

```
// Store
// Crear un par de nombre/valor de almacenamiento local con nombre="apellido" y valor="Peláez"
localStorage.setItem("apellido", "Peláez");

// Retrieve
// Recuperar el valor de "apellido" e insertarlo en el elemento con id="resultado"
document.getElementById("resultado").innerHTML = localStorage.getItem("apellido");
```

El ejemplo anterior también podría escribirse así:

```
// Store
localStorage.apellido = "Peláez";
// Retrieve
document.getElementById("resultado").innerHTML = localStorage.apellido;
```

## Eliminar información

Se utiliza el método `localStorage.removeItem()`. La sintaxis para eliminar el elemento `localStorage "apellido"` es la siguiente:

```
localStorage.removeItem("apellido");
```

## Eliminar todos los datos

Para eliminar todos los datos y dejar limpio el almacenamiento local de nuestro dominio y protocolo (origen):

```
localStorage.clear();
```

## Ejemplo:

Ejemplo que cuenta el número de veces que un usuario ha hecho clic en un botón. En este código, la cadena de valor se convierte a Number para poder incrementar el contador:

```
if (localStorage.clickcount) {
  localStorage.clickcount = Number(localStorage.clickcount) + 1;
} else {
  localStorage.clickcount = 1;
}

document.getElementById("resultado").innerHTML = "You have clicked the button " +
localStorage.clickcount + " time(s).";
```

## El objeto de almacenamiento (sessionStorage)

El objeto `sessionStorage` es igual al objeto `localStorage`, excepto que almacena los datos para una sola sesión. Los datos se eliminan cuando el usuario cierra la pestaña específica del navegador.

Se utilizan los mismos métodos que en el objeto localStorage.

### Ejemplo:

El siguiente ejemplo cuenta la cantidad de veces que un usuario ha hecho clic en un botón en la sesión actual:

```
if (sessionStorage.clickcount) {
    sessionStorage.clickcount = Number(sessionStorage.clickcount) + 1;
} else {
    sessionStorage.clickcount = 1;
}
document.getElementById("resultado").innerHTML = "You have clicked the button " +
sessionStorage.clickcount + " time(s) in this session.";
```

Ejercicio: escribe el código para guardar automáticamente el contenido de un campo de texto y, si se actualiza el navegador, restaurar el contenido del campo de texto para que no se pierda lo que ya tiene escrito.

## Persistencia de la información almacenada.

Hay que tener en cuenta que, si un usuario realiza una limpieza de la caché del navegador, hará que se borren los datos almacenados con localStorage.

Si el usuario no limpia la caché, los datos se mantendrán durante mucho tiempo. En cambio, hay usuarios que tienen configurado el navegador para que la caché se limpie en cada ocasión en que cierran el navegador. En este caso la persistencia que ofrece localStorage es similar a la que ofrece sessionStorage.

No podemos confiar el funcionamiento de una aplicación web a que el usuario limpie o no limpие la caché, por tanto, deberemos seguir trabajando con datos del lado del servidor siempre que deseemos obtener una persistencia de duración indefinida.

### El evento storage

localStorage permite que se reconozcan datos desde distintas ventanas.

Para detectar que en una ventana que se ha producido un cambio en los datos se definió el evento **storage**: este evento se dispara cuando tiene lugar un cambio en el espacio de almacenamiento y puede ser detectado por las distintas ventanas que estén abiertas.

Para crear una respuesta a este evento podemos escribir:

```
window.addEventListener("storage", nombreFuncionRespuesta, false);
```

Donde nombreFuncionRespuesta es el nombre de la función que se invocará cuando se produzca el evento.

## Diferencias entre cookies y storage

Los objetos storage juegan un papel similar a las cookies, pero por otro lado hay diferencias importantes:

- Las cookies están disponibles tanto en el servidor como en el navegador del usuario. Los objetos storage sólo están disponibles en el navegador del usuario.

- Las cookies se concibieron como pequeños paquetes de identificación, con una capacidad limitada (unos 4 Kb). Los objetos storage se han concebido para almacenar datos a mayor escala (pudiendo comprender cientos o miles de datos con un espacio de almacenamiento de varios Mb).

Hay que tener en cuenta que, de una forma u otra, ni las **cookies** ni los objetos **storage** están pensados para el almacenamiento de grandes volúmenes de información, sino para la gestión de los flujos de datos propios de la navegación web.

## Ejemplo: Almacenar un array de objetos con JSON y recuperarlo

*Archivos index.html y clasePersona.js completos:*

Tenemos un archivo index.html y otro con la clase Persona

<pre>&lt;!DOCTYPE html&gt; &lt;html lang="es"&gt; &lt;head&gt;   &lt;meta charset="UTF-8"&gt;   &lt;title&gt;Document&lt;/title&gt; &lt;/head&gt; &lt;body&gt;   &lt;script type ="module" src="js/app.js"&gt;&lt;/script&gt; &lt;/body&gt; &lt;/html&gt;</pre>	<pre>export class Persona{   constructor(nombre, apellido){     this._nombre = nombre;     this._apellido = apellido;   }   get nombre(){     return this._nombre;   }   set nombre(nombre){     this._nombre = nombre;   }   get apellido(){     return this._apellido;   }   set apellido(apellido){     return this._apellido = apellido;   } }</pre>

Se trata de crear objetos de la clase Persona, guardarlos como cadena y recuperarlos como array.

Vamos a utilizar los métodos

- JSON.stringify()
- JSON.parse()

```
//Guardar el objeto personas en localStorage
localStorage.setItem('personasCadena', JSON.stringify(personas));
```

```
// recuperar de localStorage en crudo (texto sin más)
let personasAlmacenadas = localStorage.getItem('personasCadena');
```

```
// recuperar de localStorage parseando (obteniendo el array de objetos)
// let personas2 = JSON.parse(localStorage.getItem('personasCadena'));
```

```
let personas2=JSON.parse(personasAlmacenadas);
```

Archivo app.js completo:

```
import {Persona} from './clases/clasePersona.js';

// (1) personas: es un array de objetos de la clase persona
// (2) personasAlmacenadas: es una cadena de texto para poder guardar en localStorage
// (3) personas2: es un array de objetos que conseguimos al parsear personasAlmacenadas

let personas=[
    new Persona('Juan', 'Pérez'),
    new Persona('Ana', 'González'),
    new Persona('Faustino', 'Sigüenza')
];

// Tenemos 3 personas y añadimos una cuarta
personas.push(new Persona('Andrés', 'Hernández'));
console.log('personas: ', personas);

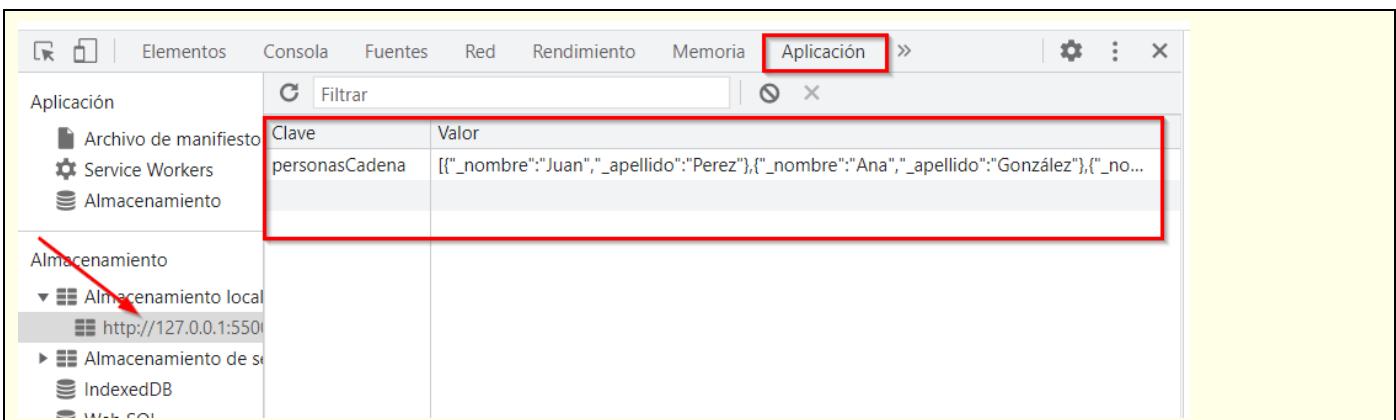
// Guardamos el objeto personas en localStorage
localStorage.setItem('personasCadena', JSON.stringify(personas));

// Recuperamos de localStorage en crudo, como texto
let personasAlmacenadas = localStorage.getItem('personasCadena');
console.log("personasAlmacenadas: ", personasAlmacenadas);

// Recuperamos de localStorage parseando
//let personas2 = JSON.parse(localStorage.getItem('personasCadena'))
let personas2 = JSON.parse(personasAlmacenadas);
console.log('personas2: ', personas2);
```

Inspeccionando el almacenamiento en localStorage:

Se puede visualizar, editar y eliminar el contenido de localStorage inspeccionando el documento:



The screenshot shows the Chrome DevTools interface with the 'Aplicación' (Application) tab selected. In the left sidebar under 'Almacenamiento', 'Almacenamiento local' is expanded, showing 'http://127.0.0.1:550'. In the main pane, there is a table with two columns: 'Clave' (Key) and 'Valor' (Value). A single row is present, with the key 'personasCadena' and the value being a JSON string representing an array of objects: '[{"\_nombre": "Juan", "\_apellido": "Perez"}, {"\_nombre": "Ana", "\_apellido": "González"}, {"\_nombre": "Faustino", "\_apellido": "Sigüenza"}]'. This row is highlighted with a red rectangle.

Clave	Valor
personasCadena	[{"_nombre": "Juan", "_apellido": "Perez"}, {"_nombre": "Ana", "_apellido": "González"}, {"_nombre": "Faustino", "_apellido": "Sigüenza"}]

## Mirando en la consola:

```

js > JS prueba.js > ...
1 // (1) personas: es un Array de objetos de la clase Persona
2 // (2) personasAlmacenadas: es una cadena de texto para poder guardar en localStorage
3 // (3) personas2: es un Array de objetos que conseguimos al parsear personasAlmacenadas
4 let personas=[
5     new Persona('Juan', 'Perez'),
6     new Persona('Ana', 'González'),
7     new Persona('Faustino', 'Sigüenza')
8 ];
9 // tenemos 3 personas y añadimos una cuarta persona
10 personas.push(new Persona('Andrés', 'Hernández'));
11 console.log('personas: ', personas);
12
13 //Guardar el objeto personas en localStorage
14 localStorage.setItem('personasCadena',JSON.stringify(personas));
15
16 // recuperar de localStorage en crudo
17 let personasAlmacenadas = localStorage.getItem('personasCadena');
18 console.log("personasAlmacenadas:", personasAlmacenadas);
19
20 // recuperar de localStorage parseando
21 //let personas2 = JSON.parse(localStorage.getItem('personasCadena'));
22 let personas2=JSON.parse(personasAlmacenadas);
23 console.log('personas2:', personas2);

```

The screenshot shows the browser's developer tools with the "Consola" tab selected. The code is run in the console, and the output is displayed in three sections. Section 1 shows the original array of Person objects. Section 2 shows the string representation of the array stored in localStorage. Section 3 shows the array of Person objects reconstructed from the stored string. Red arrows point from the code lines to their corresponding outputs.

## Problemilla:

Desplegando los arrays en la consola observamos:

```

js > JS prueba.js > ...
1 // (1) personas: es un Array de objetos de la clase Persona
2 // (2) personasAlmacenadas: es una cadena de texto para poder guardar en localStorage
3 // (3) personas2: es un Array de objetos que conseguimos al parsear personasAlmacenadas
4 let personas=[
5     new Persona('Juan', 'Perez'),
6     new Persona('Ana', 'González'),
7     new Persona('Faustino', 'Sigüenza')
8 ];
9 // tenemos 3 personas y añadimos una cuarta persona
10 personas.push(new Persona('Andrés', 'Hernández'));
11 console.log('personas: ', personas);
12
13 //Guardar el objeto personas en localStorage
14 localStorage.setItem('personasCadena',JSON.stringify(personas));
15
16 // recuperar de localStorage en crudo
17 let personasAlmacenadas = localStorage.getItem('personasCadena');
18 console.log("personasAlmacenadas:", personasAlmacenadas);
19
20 // recuperar de localStorage parseando
21 //let personas2 = JSON.parse(localStorage.getItem('personasCadena'));
22 let personas2=JSON.parse(personasAlmacenadas);
23 console.log('personas2:', personas2);

```

The screenshot shows the browser's developer tools with the "Consola" tab selected. The arrays are expanded in the console. The first array (personas) is an array of Persona objects, and its elements have method definitions. The second array (personas2), which is a plain JavaScript array of objects, does not have these method definitions. Red arrows point from the code lines to the expanded arrays in the console.

Tenemos un pequeño problema:

- Guardamos un array de objetos de la clase Persona (el llamado **personas**)
- Recuperamos un array de objetos sin clase (al que hemos llamado **personas2**)

Los datos son los mismos, pero el segundo array (personas2) no puede utilizar los métodos de la clase.

```

24 //mostrando el atributo _nombre de personas con getNombre()
25 personas.forEach(element => {
26   console.log(element.getNombre());
27 });
28 );
29
30 //mostrando el atributo _nombre de personas2 con etNombre()
31 personas2.forEach(element => {
32   console.log(element.getNombre());
33 });
34
35 //mostrando el atributo _nombre de personas directamente
36 personas.forEach(element => {
37   console.log(element._nombre);
38 });
39
40 //mostrando el atributo _nombre de personas2 directamente
41 personas2.forEach(element => {
42   console.log(element._nombre);
43 });
44

```

En el ejemplo anterior no se pueden utilizar los métodos `get()` para obtener los valores de los atributos de los elementos en `personas2`, debemos obtener los atributos directamente.

## Usando `Object.create()`

Para resolver el caso anterior, podemos utilizar `Object.create(objetoModelo)` para añadir a un array `personas3` los objetos de la clase Persona que hemos recuperado de `localStorage`.

A continuación, ponemos un ejemplo:

```

js > prueba2.js > ...
17 const personaModelo= new Persona(); 1
18
19 const nuevaPersona = Object.create(personaModelo); 2
20 nuevaPersona._nombre='Pedro'; 3
21 nuevaPersona._apellido='Gómez'; 4
22
23 console.log(nuevaPersona.getNombre()); 5
24
25 const otraPersona = Object.create(personaModelo); 6
26 otraPersona._nombre='Carmen'; 7
27 otraPersona._apellido='Sevilla'; 8
28
29 personas.push(nuevaPersona);
30 personas.push(otraPersona);
31 console.log(personas);

```

- (1) Creamos un objeto modelo de la clase que queramos tener. No hace falta que tenga datos.
- (2) Creamos un nuevo objeto utilizando el objeto modelo existente.
- (3) (6) Asignamos contenido a los atributos.
- (4) Ya podemos utilizar los métodos de la clase.
- (5) Podemos crear más objetos de la clase.
- (7) Metemos los objetos de la clase Persona en el array personas
- (8) Comprobamos el contenido del array.

## Obteniendo el array de objetos de la clase Persona:

Se va a resolver el problemilla:

*Archivo app.js resultante:*

```
import {Persona} from './clases/clasePersona.js';
```

```

// (1) personas: es un array de objetos de la clase persona
// (2) personasAlmacenadas: es una cadena de texto para poder guardar en localStorage
// (3) personas2: es un array de objetos que conseguimos al parsear personasAlmacenadas

let personas=[  

    new Persona('Juan', 'Pérez'),  

    new Persona('Ana', 'González'),  

    new Persona('Faustino', 'Sigüenza')  

];  
  

// Tenemos 3 personas y añadimos una cuarta  

personas.push(new Persona('Andrés', 'Hernández'));  

console.log('personas: ', personas);  
  

// Guardamos el objeto personas en localStorage  

localStorage.setItem('personasCadena', JSON.stringify(personas));  
  

// Recuperamos de localStorage en crudo, como texto  

let personasAlmacenadas = localStorage.getItem('personasCadena');  

console.log("personasAlmacenadas: ", personasAlmacenadas);  
  

// Recuperamos de localStorage parseando  

//let personas2 = JSON.parse(localStorage.getItem('personasCadena'))  

let personas2 = JSON.parse(personasAlmacenadas);  

console.log('personas2: ', personas2);  
  

// Recuperar el array de objetos de la clase Persona:  

// const personaModelo = new Persona(); // (**)
const personas3=[];  
  

personas2.forEach((p) =>{  

    // let nuevaPersona= Object.create(personaModelo); // (**)  

    let nuevaPersona=new Persona() // (**)  

    nuevaPersona.nombre = p._nombre;  

    nuevaPersona.apellido = p.apellido;  

    console.log(nuevaPersona);  

    personas3.push(nuevaPersona);  

})  
  

console.log('personas3: ',personas3);

```

(\*\*) Se puede optar por:

- a) Utilizar personaModelo como modelo de nuevaPersona, o
- b) Crear nuevaPersona como una instancia de la clase Persona.



## DWEC - Javascript Web Cliente.

JavaScript - Ajax 1 .....	1
Introducción.....	1
Métodos HTTP.....	3
Formato JSON .....	4
Json Server .....	7
REST client.....	10
Thundert Client para VSC y json-server.....	13

# JavaScript - Ajax 1

## Introducción

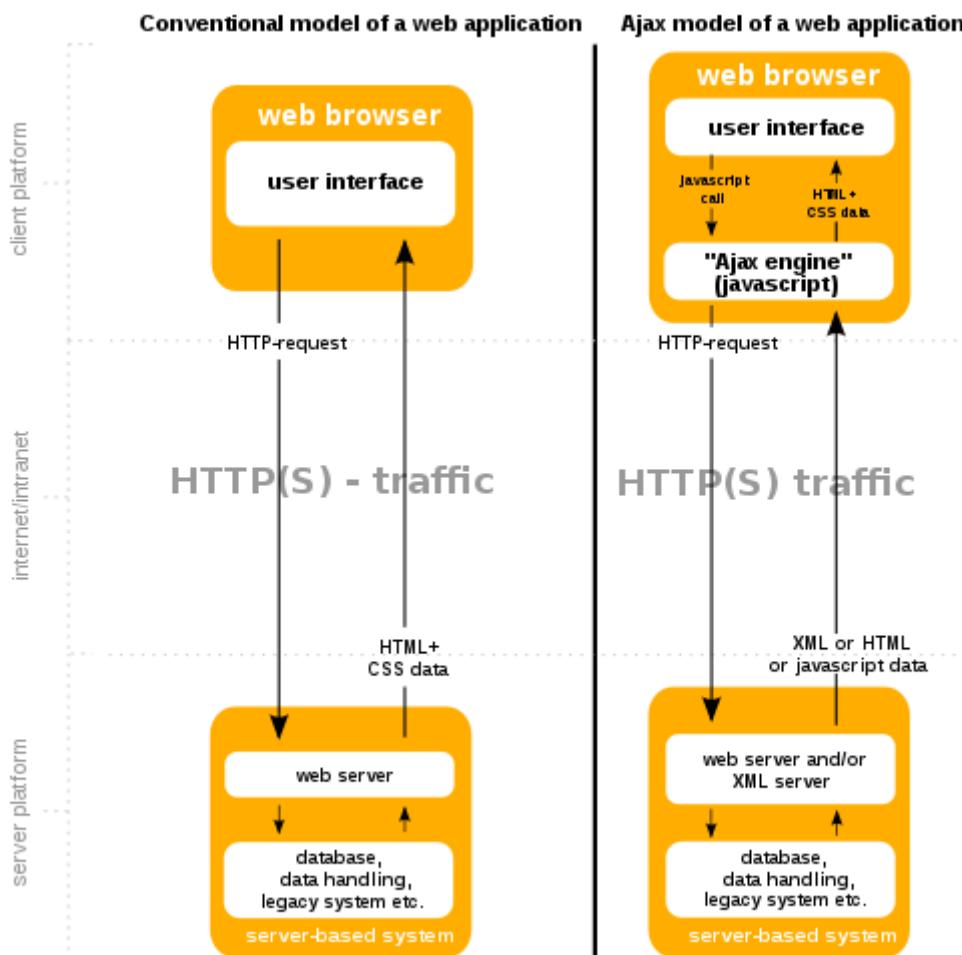
AJAX es el acrónimo de **Asynchronous Javascript And XML** (Javascript asíncrono y XML) y es lo que usamos para hacer peticiones asíncronas al servidor desde Javascript.

Cuando hacemos una petición al servidor no nos responde inmediatamente (la petición tiene que llegar al servidor, procesarse allí y enviarse la respuesta que llegará al cliente).

Lo que significa **asíncrono** es que la página (cliente) no permanecerá bloqueada esperando esa respuesta, sino que continuará ejecutando su código e interactuando con el usuario y, en el momento en que llegue la respuesta del servidor se ejecutará la función que habíamos indicado al hacer la llamada Ajax.

Respecto a **XML**: es el formato en que se intercambia la información entre el servidor y el cliente, aunque actualmente el formato más usado es **JSON** que es más simple y legible. Podríamos decir que hoy no se usa Ajax y sí Ajaj, pero suena a que te pica la garganta, así que seguiremos diciendo Ajax aunque usemos JSON.

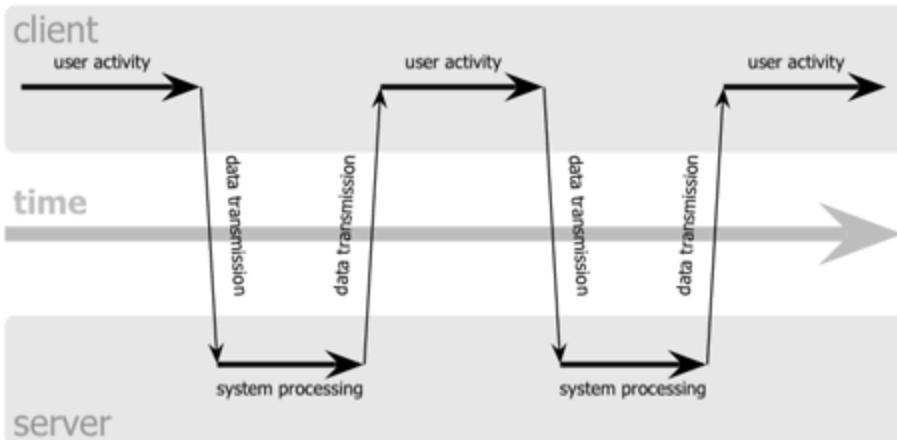
Básicamente Ajax nos permite poder mostrar nuevos datos enviados por el servidor sin tener que recargar la página, que continuará disponible mientras se reciben y procesan los datos enviados por el servidor en segundo plano.



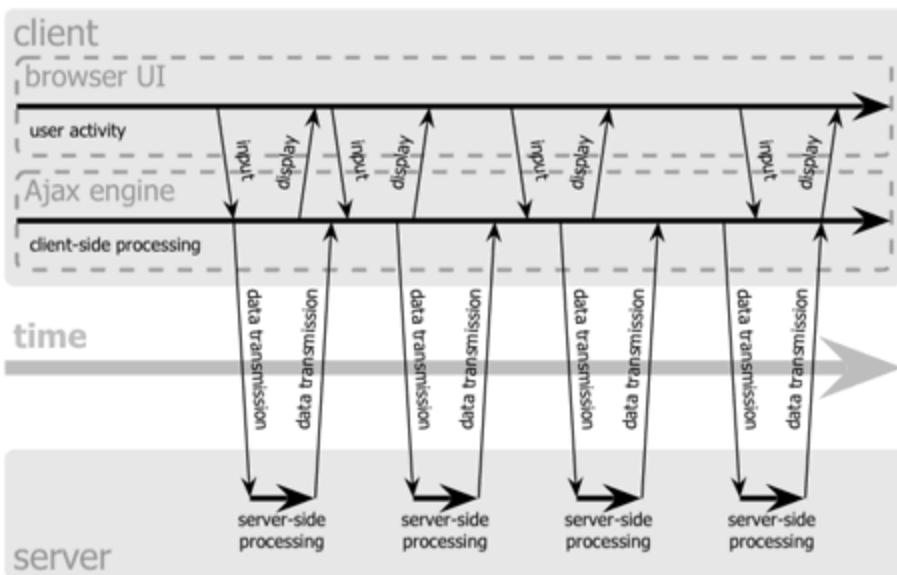
Sin Ajax, cada vez que necesitamos nuevos datos del servidor la página deja de estar disponible para el usuario hasta que se recarga con lo que envía el servidor.

Con Ajax la página está siempre disponible para el usuario y simplemente se modifica (cambiando el DOM) cuando llegan los datos del servidor:

## classic web application model (synchronous)



## Ajax web application model (asynchronous)



## Métodos HTTP

Las peticiones Ajax usan el protocolo HTTP (el mismo que utiliza el navegador para cargar una página).

El protocolo HTTP envía al servidor:

- Unas cabeceras HTTP con información como el *userAgent* del navegador, el idioma, etc.
- El tipo de petición.
- Opcionalmente, datos o parámetros. Por ejemplo, en la petición que procesa un formulario se envían los datos del mismo.

Hay diferentes tipos de petición que podemos hacer:

- GET**: suele usarse para obtener datos sin modificar nada (equivale a un SELECT en SQL). Si enviamos datos (ej. el ID del registro a obtener) suelen ir en la url de la petición (formato URLEncoded). Ej.: [localhost/users/3](http://localhost/users/3), <https://jsonplaceholder.typicode.com/users> o [www.google.es?search=js](http://www.google.es?search=js)
- POST**: suele usarse para añadir un dato en el servidor (equivalente a un INSERT). Los datos enviados van en el cuerpo de la petición HTTP (igual que sucede al enviar desde el navegador un formulario por POST)
- PUT**: es similar al POST, pero suele usarse para actualizar datos del servidor (como un UPDATE de SQL). Los datos se envían en el cuerpo de la petición (como en el POST). La información para identificar el objeto

a modificar va en la url (como en el GET). El servidor hará un UPDATE sustituyendo el objeto actual por el que se le pasa como parámetro.

- **PATCH**: es similar al PUT, pero la diferencia es que en el PUT hay que pasar todos los campos del objeto a modificar (los campos no pasados se eliminan del objeto), mientras que en el PATCH sólo se pasan los campos que se quieren cambiar y el resto permanecen como están.
- **DELETE**: se usa para eliminar un dato del servidor (como un DELETE de SQL). La información para identificar el objeto a eliminar se envía en la url (como en el GET).
- Existen otros tipos de peticiones que no trataremos en esta documentación, que son:
  - HEAD: pide una respuesta idéntica a la de una petición GET, pero sin el cuerpo de la respuesta.
  - CONNECT: establece un túnel hacia el servidor identificado por el recurso.
  - OPTIONS: es utilizado para describir las opciones de comunicación para el recurso de destino.
  - TRACE: realiza una prueba de bucle de retorno de mensaje a lo largo de la ruta al recurso de destino.

El servidor acepta la petición, la procesa y envía al cliente una respuesta que consiste en:

- a) El recurso solicitado.
- b) Adjunta unas cabeceras de respuesta que incluyen: el tipo de contenido enviado, el idioma, etc.
- c) El código de estado.

Los códigos de estado más comunes son:

- 2xx: son peticiones procesadas correctamente. Las más usuales son 200 (*ok*) o 201 (*created*, como respuesta a una petición POST satisfactoria).
- 3xx: son códigos de redirección que indican que la petición se redirecciona a otro recurso del servidor, como 301 (el recurso se ha movido permanentemente a otra URL) o 304 (el recurso no ha cambiado desde la última petición por lo que se puede recuperar desde la caché).
- 4xx: indican un error por parte del cliente, como 404 (*Not found*, no existe el recurso solicitado) o 401 (*Not authorized*, el cliente no está autorizado a acceder al recurso solicitado).
- 5xx: indican un error por parte del servidor. Ejemplos: 500 (error interno del servidor) o 504 (*timeout*, el servidor no responde).

En cuanto a la información enviada por el servidor al cliente, normalmente serán datos en formato **JSON** o XML (cada vez menos usado) que el cliente procesará y mostrará en la página al usuario. También podría ser HTML, texto plano, ...

## Formato JSON

El formato JSON (acrónimo de JavaScript Object Notation, 'notación de objeto de JavaScript') es una forma de convertir objetos Javascript en una cadena de texto para poderlos enviar.

La estructura de datos JSON se compone de un conjunto de objetos o arrays que contendrán números, cadenas booleanos y nulos.

Un **objeto** JSON comienza y termina con llaves, y contiene una colección desordenada de pares **nombre-valor**.

Cada **nombre** y **valor** están separados por dos puntos, y los pares están separados por comas.

La coma final está prohibida.

El **nombre** es una cadena entre comillas dobles. Los caracteres de comillas no deben ser inclinadas o "inteligentes".

En los **números**, los ceros a la izquierda están prohibidos; un punto decimal debe estar seguido al menos por un dígito.

En las cadenas deben estar entre comillas dobles. No se permiten todos los caracteres de escape; sí se permiten los caracteres de separador de línea Unicode (U+2028) y el separador de párrafo (U+2029).

Un array JSON comienza y termina con corchetes y contiene una colección ordenada de valores separados por comas. Un valor puede ser una cadena entre comillas dobles, un número, un booleano true o false, nulo, un objeto JSON o un array.

Los objetos y los arrays JSON se pueden anidar, lo que posibilita una estructura jerárquica de datos.

En el siguiente ejemplo, se muestra una estructura de datos JSON con dos objetos válidos.

```
{
  "id": 1006410,
  "title": "Amazon Redshift Database Developer Guide"
}
{
  "id": 100540,
  "name": "Amazon Simple Storage Service User Guide"
}
```

En el siguiente se muestran los mismos datos como dos arrays JSON:

```
[
  1006410,
  "Amazon Redshift Database Developer Guide"
]
[
  100540,
  "Amazon Simple Storage Service User Guide"
]
```

### Conversiones de objetos

El objeto alumno:

```
let alumno = {
  id: 5,
  nombre: 'Ana',
  apellidos: 'Zubiri Peláez'
}
```

se transformaría en la cadena de texto:

```
{ "id": 5, "nombre": "Ana", "apellidos": "Zubiri Peláez" }
```

y el array:

```
let alumnos = [
  {
    id: 5,
    nombre: "Ana",
    apellidos: "Zubiri Peláez"
  },
  {
    id: 6,
    nombre: "Juan",
    apellidos: "Gómez"
  }
]
```

```
{
  id: 7,
  nombre: "Carlos",
  apellidos: "Pérez Ortíz"
},
]
```

en la cadena:

```
[{ "id": 5, "nombre": "Ana", "apellidos": "Zubiri Peláez" }, { "id": 7, "nombre": "Carlos", "apellidos": "Pérez Ortíz" }]
```

Nótese que tanto las claves como los valores van entrecomillados (con comillas dobles). No sirven comillas simples.

### Estructura de los datos

Los mismos datos pueden tener distinta estructura. Ejemplo:

Archivo colores1.json	Archivo colores2.json	Archivo colores3.json
{ "arrayColores": [ { "nombreColor": "rojo", "valorHexadec": "#f00" }, { "nombreColor": "verde", "valorHexadec": "#0f0" }, { "nombreColor": "azul", "valorHexadec": "#00f" }, { "nombreColor": "cyan", "valorHexadec": "#0ff" }, { "nombreColor": "magenta", "valorHexadec": "#f0f" }, { "nombreColor": "amarillo", "valorHexadec": "#ff0" }, { "nombreColor": "negro", "valorHexadec": "#000" } ]	{ "arrayColores": [ { "rojo": "#f00", "verde": "#0f0", "azul": "#00f", "cyan": "#0ff", "magenta": "#f0f", "amarillo": "#ff0", "negro": "#000" }] }	{ "rojo": "#f00", "verde": "#0f0", "azul": "#00f", "cyan": "#0ff", "magenta": "#f0f", "amarillo": "#ff0", "negro": "#000" }

Los ejemplos anteriores representan lo que podrían ser archivos JSON conteniendo datos en formato JSON.

Se trata de 3 archivos que contienen aproximadamente la misma información. Sin embargo, hay algunas diferencias:

- En el archivo **colores1.json** existe un único objeto de datos donde el nombre es *arrayColores* y su valor es un array de objetos JSON. Cada objeto del array está formado por los pares (*nombreColor* y su valor), y (*valorHexadec* y su valor). En este ejemplo en concreto el array consta de 7 elementos con información correspondiente a 7 colores.
- En el archivo **colores2.json** existe un único objeto de datos donde el nombre es *arrayColores*, cuyo valor es un array que contiene un único objeto JSON formado por siete pares (nombre – valor) que representa información sobre siete colores.
- En el archivo **colores3.json** existe un único objeto de datos que está formado por siete pares (nombre – valor) que representa información sobre siete colores.

Siendo las 3 formas válidas, se deberá utilizar aquella que se indique en las instrucciones o, de no existir pautas precisas, utilizar aquel diseño que favorezca el desarrollo y mantenimiento de la aplicación.

### Métodos del objeto JSON

Para convertir objetos en cadenas de texto JSON y viceversa, Javascript proporciona 2 métodos:

- **JSON.stringify(objeto)**: recibe un objeto JS y devuelve la cadena de texto correspondiente.

```
//convierte de formato JSON a objeto
const cadenaAlumnos = JSON.stringify(alumnos)
```

- **JSON.parse(cadena)**: realiza el proceso inverso, convirtiendo una cadena de texto en un objeto.

```
//convierte un objeto a formato JSON
const alumnos = JSON.parse(cadenaAlumnos)
```

## Json Server

Las peticiones Ajax se hacen a un servidor que proporcione una API.

Se puede utilizar **Json Server** que es un servidor API-REST que funciona bajo Node.js

(nota para mí: me ha fallado Node ver.18, instalé versión 16 y bien)

Si aún no está instalado Node.js, se puede hacer desde <https://nodejs.org/es/>

Node.js utiliza un fichero JSON como contenedor de los datos en lugar de una base de datos.

Para instalar json-server en nuestra máquina, lo instalaremos de forma global para poderlo utilizar en otros ejercicios, desde cualquier terminal hay que ejecutar **npm** (Node Package Manager) de la siguiente forma:

```
npm install -g json-server
```

Para que sirva (provea de este servicio los datos de) un fichero datos.json, se ejecuta la sentencia:

```
json-server datos.json
```

Se puede poner la opción **--watch** ( o **-w**) para que actualice los datos si se modifica el fichero **.json** externamente (si lo editamos).

El fichero **datos.json** será un fichero que contenga un objeto JSON con una propiedad para cada “*tabla*” de nuestra BBDD.

Por ejemplo, si queremos simular una BBDD con las tablas *users* y *posts* vacías, el contenido del fichero será:

```
{
  "users": [],
  "posts": []
}
```

Otro ejemplo de un fichero json con 2 tablas: películas y clasificaciones

Películas y clasificaciones son dos entidades distintas.

```
{
```

```
"peliculas": [
  {
    "id": 1,
    "nombre": "El sexto sentido",
    "director": "M. Night Shyamalan",
    "clasificacion": "Drama"
  },
  {
    "id": 2,
    "nombre": "Pulp Fiction",
    "director": "Tarantino",
    "clasificacion": "Acción"
  },
  {
    "id": 3,
    "nombre": "Todo Sobre Mi Madre",
    "director": "Almodobar",
    "clasificacion": "Drama"
  },
  {
    "id": 4,
    "nombre": "300",
    "director": "Zack Snyder",
    "clasificacion": "Acción"
  },
  {
    "id": 5,
    "nombre": "El silencio de los corderos",
    "director": "Jonathan Demme",
    "clasificacion": "Drama"
  },
  {
    "id": 6,
    "nombre": "Forrest Gump",
    "director": "Robert Zemeckis",
    "clasificacion": "Comedia"
  },
  {
    "id": 7,
    "nombre": "Las Hurdes",
    "director": "Luis Buñuel",
    "clasificacion": "Documental"
  }
],
"clasificaciones": [
  {
    "nombre": "Drama",
    "id": 1
  },
  {
    "nombre": "Comedia",
    "id": 2
  }
]
```

```

},
{
  "nombre": "Acción",
  "id": 4
},
{
  "nombre": "Terrorífica",
  "id": 15
}
]
}

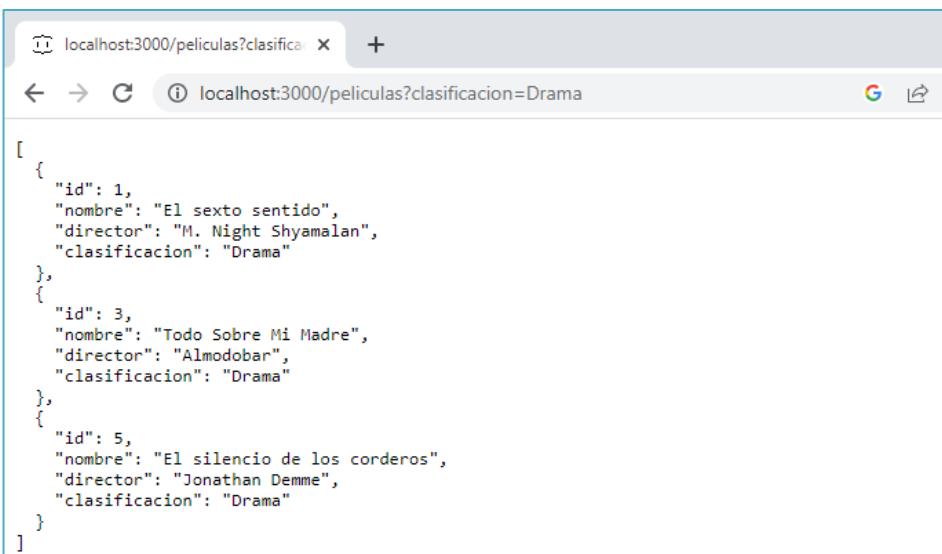
```

Al iniciar el servicio, por defecto la API escucha en el puerto 3000 y servirá los diferentes objetos definidos en el fichero `.json`. Por ejemplo:

- `http://localhost:3000/peliculas/`: devuelve un array con todos los elementos de la tabla `peliculas` del fichero `.json`
- `http://localhost:3000/peliculas/5`: devuelve un objeto con el elemento de la tabla `peliculas` cuya propiedad `id` valga 5

También pueden hacerse peticiones más complejas como:

- `http://localhost:3000/peliculas?clasificacion=Drama`: devuelve un array con todos los elementos de `peliculas` cuya propiedad `clasificacion` valga `Drama`. La siguiente imagen muestra el resultado:



The screenshot shows a browser window with the URL `localhost:3000/peliculas?clasificacion=Drama`. The page displays a JSON array of three movie objects, each with properties: id, nombre, director, and clasificacion. All three movies in the array have the value "Drama" for the "clasificacion" property.

```

[
  {
    "id": 1,
    "nombre": "El sexto sentido",
    "director": "M. Night Shyamalan",
    "clasificacion": "Drama"
  },
  {
    "id": 3,
    "nombre": "Todo Sobre Mi Madre",
    "director": "Almodobar",
    "clasificacion": "Drama"
  },
  {
    "id": 5,
    "nombre": "El silencio de los corderos",
    "director": "Jonathan Demme",
    "clasificacion": "Drama"
  }
]

```

Para más información: <https://github.com/typicode/json-server>.

Si queremos acceder a la API desde otro equipo (no desde `localhost`) tenemos que indicar la IP de la máquina que ejecuta `json-server` y que se usará para acceder. Por ejemplo, si vamos a ejecutar el servidor en la máquina 192.168.0.10 pondremos:

```
json-server --host 192.168.0.10 datos.json
```

Si se desea cambiar el puerto por defecto (3000) donde escucha el servidor, a otro (por ejemplo, el 4200):

```
json-server --host 192.168.0.10 -p 4200 datos.json
```

Y la ruta para acceder a la API será `http://192.168.0.10:4200`.

**EJERCICIO:** instalar json-server en tu máquina. Ejecútalo indicando un nombre de fichero que no existe: como verás crea un fichero json de prueba con 3 tablas: *posts*, *comments* y *profiles*. Ábrelo en tu navegador para ver los datos

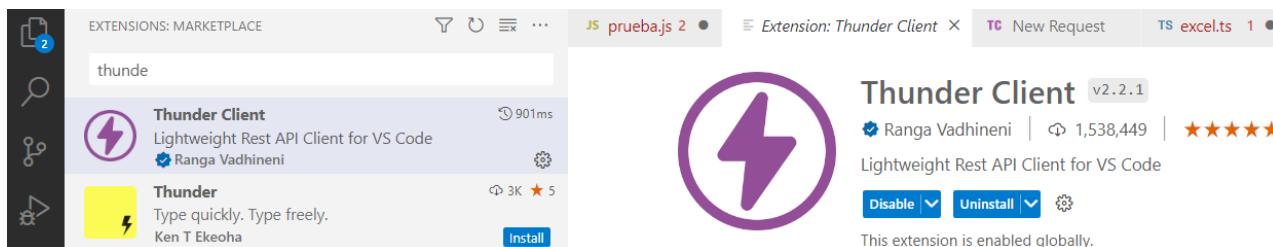
## REST client

Para probar las peticiones GET podemos poner la URL en la barra de direcciones del navegador, pero para probar el resto de peticiones debemos instalar en nuestro navegador una extensión que nos permita realizar las peticiones indicando el método a usar, las cabeceras a enviar y los datos que enviaremos a servidor, además de la URL.

Existen multitud de aplicaciones para realizar peticiones HTTP, como [Advanced REST client](#), [Postman](#), etc.

Además, cada navegador tiene sus propias extensiones para hacer esto, como [Advanced Rest Client](#) para Chrome o [RestClient](#) para Firefox.

También se puede utilizar una extensión para Visual Studio Code llamada *Thunder Client*.

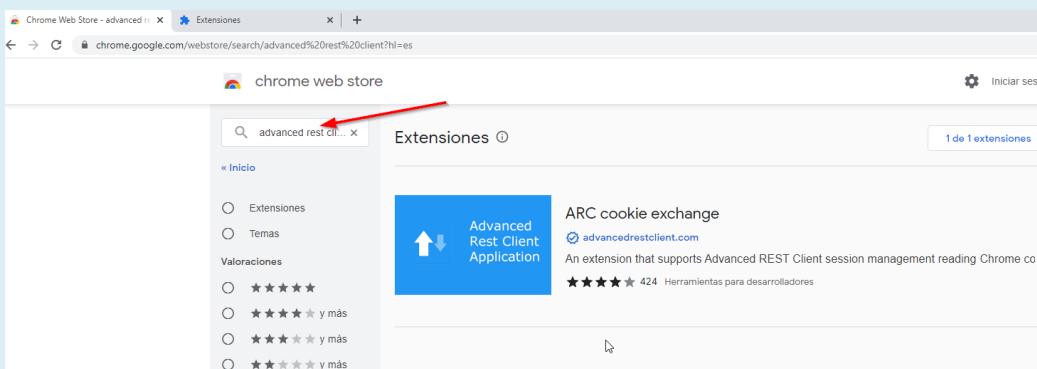


Para realizar el siguiente ejercicio hay que instalar alguna de las extensiones mencionadas y hacer todas las peticiones desde allí (incluyendo los GET), lo que permitirá ver los códigos de estado devueltos, las cabeceras, etc.

**EJERCICIO:** Vamos a realizar diferentes peticiones HTTP a la API <https://jsonplaceholder.typicode.com>. En concreto trabajaremos contra la tabla *todos* (que contiene **tareas** para hacer).

Esta API está disponible para hacer pruebas sin necesidad de montar un servidor.

Para la resolución de este ejercicio se instaló la extensión Advanced Rest Client en Google Chrome:



1º Obtener todas las tareas de la tabla todos. Devolverá un array con todas las tareas y el código devuelto será 200  
– Ok

**Request**

Method: GET Request URL: https://jsonplaceholder.typicode.com/todos

Parameters: 200 OK 198.66 ms

```
[Array[200]
-0: {
  "userId": 1,
  "id": 1,
  "title": "delectus aut autem",
  "completed": false
},
-1: {
  "userId": 1,
  "id": 2,
  "title": "quis ut nam facilis et officia qui",
  "completed": false
},
-2: {
  "userId": 1,
  "id": 3,
  "title": "fugiat veniam minus",
  "completed": false
},
-3: {
  "userId": 1,
  "id": 4,
  "title": "et porro tempora",
  "completed": true
},
-4: {
  "userId": 1,
  "id": 5,
  "title": "laboriosam ut enim et",
  "completed": false
}, ...]
```

2º Obtener la tarea cuyo id sea 55. Devolverá el objeto de la tarea 55 y el código devuelto será 200 – Ok

**Request**

Method: GET Request URL: https://jsonplaceholder.typicode.com/todos/55

Parameters: 200 OK 185.43 ms

```
{
  "userId": 3,
  "id": 55,
  "title": "voluptatum omnis minima qui occaecati provident nulla voluptatem ratione",
  "completed": true
}
```

3º Obtener la tarea con id 201. Como no existe, devolverá un objeto vacío y el código de error 404 - Not found.

**Request**

Method: GET Request URL: https://jsonplaceholder.typicode.com/todos/201

Parameters: 404 Not Found 415.52 ms

```
{}
```

4º Crear una nueva tarea. En el cuerpo de la petición le pasaremos sus datos: `userID: 1, title: Prueba de POST y completed: false`. No se le pasa la id (de eso se encarga la BBDD, que la creará automática). La respuesta debe ser un código 201 (created) y adjuntará el nuevo registro creado con todos sus datos incluyendo la id. Como es una API de prueba en realidad no lo está añadiendo a la BBDD, por lo que si luego hacemos una petición buscando esa id, nos dirá que no existe.

Advanced REST client

Method: POST Request URL: https://jsonplaceholder.typicode.com/todos

Parameters ^

Headers Body Variables

Body content type application/json Editor view Raw input

FORMAT JSON MINIFY JSON

```
{
  "userId": 1,
  "title": "Prueba de POST",
  "completed": false
}
```

201 Created 583.57 ms DETAILS ▾

□ ☰ <> ■■■

```
{
  "userId": 1,
  "title": "Prueba de POST",
  "completed": false,
  "id": 201
}
```

5º Modificar con un PATCH la tarea con id 55 para que su title sea ‘Prueba de POST’. Devolverá el nuevo registro con un código 200. Se observa que al hacer un PATCH los campos que no se pasan se mantienen como estaban.

Advanced REST client

Method: PATCH Request URL: https://jsonplaceholder.typicode.com/todos/55

Parameters ^

Headers Body Variables

Body content type application/json Editor view Raw input

FORMAT JSON MINIFY JSON

```
{
  "title": "Prueba de POST"
}
```

200 OK 381.48 ms DETAILS ▾

□ ☰ <> ■■■

```
{
  "userId": 3,
  "id": 55,
  "title": "Prueba de POST",
  "completed": true
}
```

6º Modificar con un PUT la tarea con id 55 para que su title sea ‘Prueba de POST’. Devolverá el nuevo registro con un código 200. Como se aprecia en la imagen, en esta API los campos que no se pasan se eliminan; en otras los campos no pasados se mantienen como estaban.

Advanced REST client

### Request

Method: PUT Request URL: https://jsonplaceholder.typicode.com/todos/55

SEND ⋮

Parameters ^

Headers	Body	Variables
Body content type: application/json	Editor view Raw input	

FORMAT JSON MINIFY JSON

```
{
  "title": "Prueba de POST"
}
```

200 OK 417.70 ms DETAILS ▾

```
{
  "title": "Prueba de POST",
  "id": 55
}
```

7º Eliminar con DELETE la tarea con id 55. Como se ve en la siguiente imagen, esta API devuelve un objeto vacío al eliminar; otras devuelven el objeto eliminado.

Advanced REST client

### Request

Method: DELETE Request URL: https://jsonplaceholder.typicode.com/todos/55

SEND ⋮

Parameters ^

Headers	Body	Variables
Body content type: application/json	Editor view Raw input	

FORMAT JSON MINIFY JSON

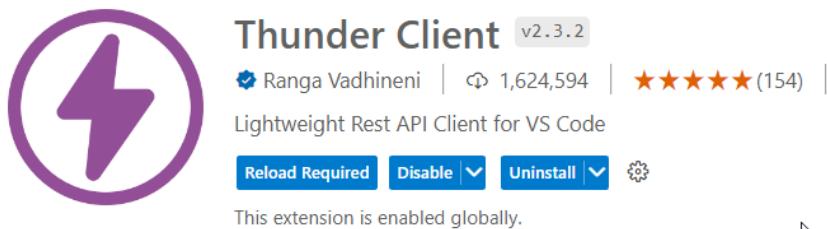
```
*
```

200 OK 499.48 ms DETAILS ▾

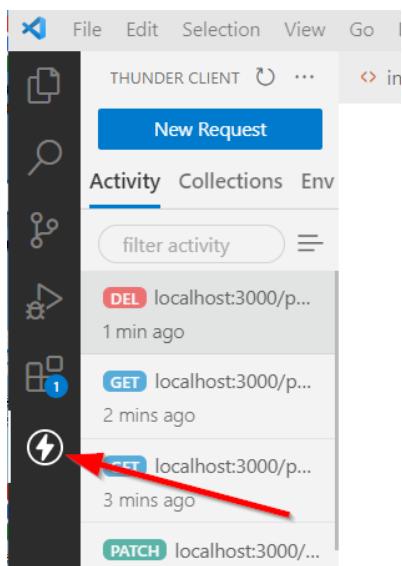
```
{}
```

## Thundert Client para VSC y json-server

Se instala como una versión de Visual Studio Code



Una vez instalada nos proporciona en VSC una interfaz cómoda para realizar peticiones.



Para trabajar con json-server, podemos utilizar un terminal de VSC, tanto para instalar json-server:

```
PROBLEMS OUTPUT TERMINAL powershell
PS C:\Users\Santi\DAW2cliente\CURSO_22_23\AJAX\thunder1> npm install json-server
```

como para servir los datos de un fichero json:

```
PS C:\Users\Santi\DAW2cliente\CURSO_22_23\AJAX\POST01> json-server.cmd .\peliculas.json
\{^_^\}/ hi!
Loading .\peliculas.json
Done

Resources
http://localhost:3000/peliculas
http://localhost:3000/clasificaciones
```

Si en algún momento en el terminal nos apareciera un mensaje advirtiendo que no se pueden ejecutar scripts, podemos desde el mismo terminal o desde un terminal PowerShell activar la política que deseemos sobre la ejecución de scripts. Se recomienda el modo RemoteSigned.

```
Administrator: Windows PowerShell
Windows PowerShell
Copyright (C) Microsoft Corporation. Todos los derechos reservados.

Prueba la nueva tecnología PowerShell multiplataforma https://aka.ms/pscore6

PS C:\WINDOWS\system32> Get-ExecutionPolicy
Restricted
PS C:\WINDOWS\system32> Set-ExecutionPolicy RemoteSigned
```



## DWEC - Javascript Web Cliente.

JavaScript - Ajax 2 .....	1
Realizar peticiones Ajax .....	1
Eventos de XMLHttpRequest .....	2
Ejemplos de envío de datos .....	4
Enviar datos al servidor en formato JSON.....	5
Enviar datos al servidor en formato URLEncoderD .....	5
Enviar ficheros al servidor con FormData .....	6

# JavaScript - Ajax 2

## Realizar peticiones Ajax

Hemos visto lo que es el protocolo HTTP. Ahora que tenemos instalado un servidor que nos proporciona una API (json-server), vamos a realizar peticiones HTTP en nuestro código javascript usando Ajax.

Para hacer una petición debemos crear una instancia del objeto **XMLHttpRequest** que es el que controlará todo el proceso. Los pasos a seguir son:

1. Creamos la instancia del objeto: `const peticion=new XMLHttpRequest()`
2. Para establecer la comunicación con el servidor ejecutamos el método **.open()** al que se le pasa como parámetro el tipo de petición (GET, POST, ...) y la URL del servidor: `peticion.open('GET', 'https://jsonplaceholder.typicode.com/todos')`
3. OPCIONAL: Si queremos añadir cabeceras a la petición HTTP llamaremos al método **.setRequestHeader()**. Por ejemplo, si enviamos datos con POST hay que añadir la cabecera `Content-type` que indica al servidor en qué formato van los datos: `peticion.setRequestHeader('Content-type', 'application/x-www-form-urlencoded')`
4. Enviamos la petición al servidor con el método **.send()**. A este método se le pasa como parámetro los datos a enviar al servidor en el cuerpo de la petición (si es un POST, PUT o PATCH le pasaremos una cadena de texto con los datos a enviar:

```
peticion.send('dato1='+encodeURIComponent(dato1)+'&dato2='+encodeURIComponent(dato2))
```

Si es una petición GET o DELETE no le pasaremos datos:

```
peticion.send()
```

1. Ponemos un escuchador (*listener*) al objeto `peticion` para saber cuándo está disponible la respuesta del servidor.

## Eventos de XMLHttpRequest

Tenemos diferentes eventos que el servidor envía para informarnos del estado de nuestra petición y que nosotros podemos capturar:

El evento **readystatechange** se produce cada vez que el servidor cambia el estado de la petición.

Cuando hay un cambio en el estado cambia el valor de la propiedad **readyState** de la petición. Sus valores posibles son:

0: petición no iniciada (se ha creado el objeto XMLHttpRequest)

1: establecida conexión con el servidor (se ha hecho el *open*)

2: petición recibida por el servidor (se ha hecho el *send*)

3: se está procesando la petición

4: petición finalizada y respuesta lista (este es el evento que nos interesa porque ahora tenemos la respuesta disponible). A nosotros sólo nos interesa cuando su valor sea 4 que significa que ya están los datos. En ese momento la propiedad **status** contiene el estado de la petición HTTP (200: *Ok*, 404: *Not found*, 500: *Server error*, ...) que ha devuelto el servidor.

Cuando *readyState* vale 4 y *status* vale 200: tenemos los datos en la propiedad **responseText** (o **responseXML** si el servidor los envía en formato XML).

Ejemplo:

```
const peticion = new XMLHttpRequest();
console.log("Estado inicial de la petición: " + peticion.readyState);
peticion.open('GET', 'https://jsonplaceholder.typicode.com/users');
console.log("Estado de la petición tras el 'open': " + peticion.readyState);
peticion.send();
console.log("Petición hecha");
peticion.addEventListener('readystatechange', function() {
    console.log("Estado de la petición: " + peticion.readyState);
    if (peticion.readyState === 4) {
        if (peticion.status === 200) {
            console.log("Datos recibidos:");
            let usuarios = JSON.parse(peticion.responseText); // Pasamos los datos JSON a un objeto
            console.log(usuarios);
        } else {
            console.log("Error " + peticion.status + " (" + peticion.statusText + ") en la petición");
        }
    }
})
console.log("Petición acabada");
```

El resultado de ejecutar ese código es el siguiente:

The screenshot shows two panels. On the left is a code editor with a file named 'ajax.js' containing the following code:

```

1 const peticion = new XMLHttpRequest();
2 console.log("Estado inicial de la petición: " + peticion.readyState);
3 peticion.open('GET', 'https://jsonplaceholder.typicode.com/users');
4 console.log("Estado de la petición tras el 'open': " + peticion.readyState);
5 peticion.send();
6 console.log("Petición hecha");
7 peticion.addEventListener('readystatechange', function() {
8   console.log("Estado de la petición: " + peticion.readyState);
9   if (peticion.readyState === 4) {
10     if (peticion.status === 200) {
11       console.log("Datos recibidos:");
12       let usuarios = JSON.parse(peticion.responseText); // Convertimos los datos JSON a un objeto
13       console.log(usuarios);
14     } else {
15       console.log("Error " + peticion.status + " (" + peticion.statusText + ") en la petición");
16     }
17   }
18 })
19 console.log("Petición acabada");
20

```

On the right is the Chrome DevTools Console tab, showing the output of the code. It lists several log statements:

- Estado inicial de la petición: 0
- Estado de la petición tras el 'open': 1
- Petición hecha
- Petición acabada
- Estado de la petición: 2
- Estado de la petición: 3
- Estado de la petición: 4
- Datos recibidos:
- (10) [{} , {} , {} , {} , {} , {} , {} , {} , {} , {} ]

Nótese que cuando cambia de estado la petición, `readyState` cambia de valor.

- `readyState` vale 0 al crear el objeto XMLHttpRequest
- `readyState` vale 1 cuando abrimos la conexión con el servidor
- Luego se envía al servidor y es éste el que va informando al cliente de cuándo cambia el estado

**MUY IMPORTANTE:** La última línea ('Petición acabada') se ejecuta antes que las de 'Estado de la petición'. Ocurre así porque es una **petición asíncrona** y la ejecución del programa continúa sin esperar a que responda el servidor.

Como normalmente no nos interesa saber cada cambio en el estado de la petición, sino que sólo queremos saber cuándo ha terminado de procesarse, tenemos otros **eventos** que nos pueden ser de utilidad:

- **load:** se produce cuando se recibe la respuesta del servidor. Equivale a `readyState==4`. En `status` tendremos el estado de la respuesta
- **error:** se produce si sucede algún error al procesar la petición (de red, de servidor, ...)
- **timeout:** si ha transcurrido el tiempo indicado y no se ha recibido respuesta del servidor. Se puede cambiar el tiempo por defecto modificando la propiedad `timeout` antes de enviar la petición.
- **abort:** si se cancela la petición (se hace llamando al método `.abort()` de la petición).
- **loadend:** se produce siempre que termina la petición, independientemente de si se recibe respuesta o sucede algún error (incluyendo un `timeout` o un `abort`).

Ejemplo de código que sí usaremos:

```

const peticion=new XMLHttpRequest();
peticion.open('GET', 'https://jsonplaceholder.typicode.com/users');
peticion.send();
peticion.addEventListener('load', function() {
  if (peticion.status==200) {
    let usuarios=JSON.parse(peticion.responseText);
    // procesamos los datos que tenemos en usuarios
    console.log('correcto:');
    console.log(usuarios);
  } else {
    muestraError(peticion);
  }
})
peticion.addEventListener('error', muestraError);
peticion.addEventListener('abort', muestraError);
peticion.addEventListener('timeout', muestraError);

function muestraError(peticion) {

```

```

if (peticion.status) {
    console.log("Error "+peticion.status+" ("+peticion.statusText+) en la petición");
} else {
    console.log("Ocurrió un error o se abortó la conexión");
}
}

```

Conviene recordar que tratamos con peticiones asíncronas por lo que tras la línea:

```
peticion.addEventListener('load', function() {
```

no se ejecuta la línea siguiente:

```
if (peticion.status==200) {
```

sino la de:

```
peticion.addEventListener('error', muestraError);
```

Una petición asíncrona es como pedir una pizza: tras encargarla por teléfono, lo siguiente no es ir a la puerta a recogerla, sino que seguimos haciendo cosas por casa y cuando suena el timbre de casa entonces vamos a la puerta a por ella.

Prueba el ejemplo anterior con la petición correcta (tal como está) y cambiando petición.open() para que se produzca algún error.

Prueba a hacer peticiones válidas y erróneas con el fichero *peliculas.json* en json-server. Prueba con el servidor json-server apagado.

### Ejemplo:

Realiza peticiones petición GET al archivo de datos “*peliculas.json*” utilizando el servior **json-server** que estará a la escucha de peticiones en el puerto 4000 de tu equipo.

- a) Una petición debe devolver las películas cuyo director sea Tarantino. Visualiza el resultado en consola.
- b) Realizar otra petición GET errónea para visualizar el error.
- c) Realiza una petición válida con el servidor apagado.

## Ejemplos de envío de datos

Vamos a ver algunos ejemplos de envío de datos al servidor con POST. Supondremos que tenemos una página con un formulario para dar de alta nuevos productos:

El fichero **productos.json** de partida puede ser el siguiente:

```
{
  "productos": [
    {
      "id": 1,
      "name": "Teclado",
      "descrip": "Teclado mecánico Cherry ps/2"
    }
  ]
}
```

Index.html sería:

```
<form id="addProduct">
    <label for="name">Nombre: </label><input type="text" name="name" id="name" required><br>
    <label for="descrip">Descripción: </label><input type="text" name="descrip" id="descrip" required><br>

    <button type="submit">Añadir</button>
</form>
```

## Enviar datos al servidor en formato JSON

```
document.getElementById('addProduct').addEventListener('submit', (event) => {
    const newProduct = {
        name: document.getElementById("name").value,
        descrip: document.getElementById("descrip").value,
    }
    const peticion = new XMLHttpRequest();
    peticion.open('POST', 'http://localhost:4000/productos');
    peticion.setRequestHeader('Content-type', 'application/json'); // Siempre tiene
                                                                // que estar esta línea si se envían datos
    peticion.send(JSON.stringify(newProduct)); // Hay que convertir el objeto
                                                // a una cadena de texto JSON para enviarlo
    peticion.addEventListener('load', function() {
        // procesamos los datos
    })
})
```

Para enviar el objeto hay que convertirlo a una cadena JSON con la función **JSON.stringify()** (es la opuesta a **JSON.parse()**). Y siempre que se envían datos al servidor hay que indicar el formato que tienen en la cabecera de *Content-type*:

```
peticion.setRequestHeader('Content-type', 'application/json');
```

**Para evitar problemas:** Ejecutar la página en el navegador después de iniciar el servicio json-server en el servidor.

## Enviar datos al servidor en formato URLEncoded

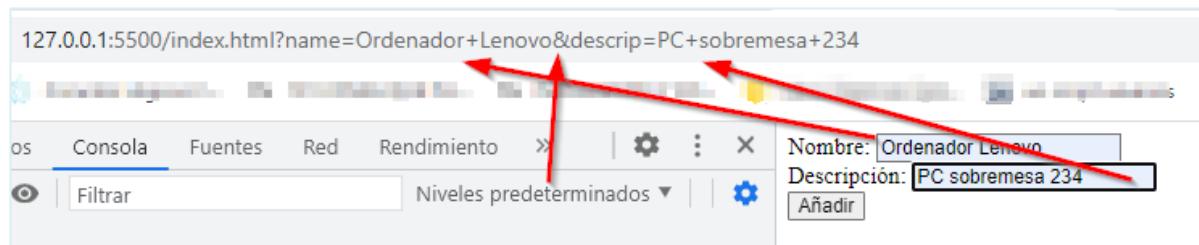
Con el mismo archivo productos.json y el mismo index.html, cambiamos el .js

```
document.getElementById('addProduct').addEventListener('submit', (event) => {
    // Aquí va el código para comprobar que los datos son correctos
    const name = document.getElementById("name").value;
    const descrip = document.getElementById("descrip").value;

    const peticion = new XMLHttpRequest();
    peticion.open('POST', 'http://localhost:4000/productos');
    peticion.setRequestHeader('Content-Type', 'application/x-www-form-urlencoded');// formato
    // la siguiente línea lleva & para separar los parámetros
    peticion.send('name=' + encodeURIComponent(name) + '&descrip=' + encodeURIComponent(descrip));
    peticion.addEventListener('load', function() {
        // procesamos los datos
    })
})
```

```
    })
})
```

En este caso los datos se envían como hace el navegador por defecto en un formulario. Recordad siempre codificar lo que introduce el usuario para evitar problemas con caracteres no estándar y **ataques SQL Injection**.



## Enviar ficheros al servidor con FormData

FormData es una interfaz de XMLHttpRequest que permite construir fácilmente pares de clave=valor para enviar los datos de un formulario. Se envían en el mismo formato en que se enviarían directamente desde un formulario (“multipart/form-data”) por lo que no hay que poner encabezado de ‘Content-type’.

Vamos a añadir al formulario un campo donde el usuario pueda subir la foto del producto:

```
<form id="addProduct">
  <label for="name">Nombre: </label><input type="text" name="name" id="name" required><br>
  <label for="descrip">Descripción: </label><input type="text" name="descrip" id="descrip" required><br>
  <label for="photo">Fotografía: </label><input type="file" name="photo" id="photo" required><br>

  <button type="submit">Añadir</button>
</form>
```

Podemos enviar al servidor todo el contenido del formulario:

```
document.getElementById('addProduct').addEventListener('submit', (event) => {
  const peticion=new XMLHttpRequest();
  const datosForm = new FormData(document.getElementById('addProduct'));
  // Automáticamente ha añadido todos los inputs, incluyendo tipo 'file', blob, ...
  // Si quisiéramos añadir algún dato más haríamos:
  formData.append('otrodato', 12345);
  // Y lo enviamos
  peticion.open('POST', 'https://localhost/products');
  peticion.send(datosForm);
  peticion.addEventListener('load', function() {
    // procesamos los datos aquí
  })
})
```

También podemos enviar sólo los campos que queramos:

```
document.getElementById('addProduct').addEventListener('submit', (event) => {
```

```
const formData=new FormData(); // creamos un formData vacío
formData.append('name', document.getElementById('name').value);
formData.append('descrip', document.getElementById('descrip').value);
formData.append('photo', document.getElementById('photo').files[0]);

const peticion=new XMLHttpRequest();
peticion.open('POST', 'https://localhost/products');
peticion.send(formData);
peticion.addEventListener('load', function() {
    // procesamos los datos aquí
})
})
```

Más información de cómo usar formData en [MDN web docs](#).



## DWEC - Javascript Web Cliente.

JavaScript - Ajax 3 .....	1
Problema con las peticiones Ajax y cómo resolverlo:.....	1
Formulario para obtener (GET) un producto dado su id:.....	3
Funciones callback .....	4
Promesas.....	5
Fetch .....	8
Propiedades y métodos de la respuesta .....	9
Gestión de errores con fetch.....	10
Otros métodos de petición POST, PUT.....	11

# JavaScript - Ajax 3

## Problema con las peticiones Ajax y cómo resolverlo:

Vamos a ver un ejemplo de una llamada a Ajax. Vamos a hacer una página que muestre en un párrafo el nombre y la descripción del producto del cual indiquemos su id en un input. En resumen, lo que hacemos es:

1. El usuario de nuestra aplicación introduce el código (id) del producto del que queremos ver sus datos.
2. Tenemos un escuchador para que al introducir un código de un usuario llamamos a una función `getProd()` que:
  - Se encarga de hacer la petición Ajax al servidor
  - Si se produce un error se encarga de informar al usuario de nuestra aplicación
3. Cuando se reciben los datos deben pintarse en la tabla

El archivo **productos.json** que contiene los datos:

```
{
  "productos": [
    {
      "id": 1,
      "name": "Teclado",
      "descrip": "Teclado mecánico Cherry ps/2"
    }
  ]
}
```

La página html sería algo así:

```
<form id="addProduct">
  <label for="id-prod">Id Producto: </label><input type="number" name="id-prod" id="id-prod">
  <br>
```

```

<button type="submit">Buscar</button>
<p id="p1">Aquí vendrán los datos</p>
</form>

```

Si Ajax no fuera una petición asíncrona el código Javascript de todo esto sería algo como el siguiente (ATENCIÓN, este código **NO FUNCIONA**):

```

1  const SERVER = 'http://localhost:4000';
2
3 window.addEventListener('load', function() {
4     document.getElementById('addProduct').addEventListener('submit', (event) => {
5         event.preventDefault();
6         let idProd = document.getElementById('id-prod').value;
7         if (isNaN(idProd) || idProd == '') {
8             alert('Debes introducir un número');
9         } else {
10            const datos = getProd(idProd);
11            console.log(datos);
12            // pintamos los datos en la página
13            document.getElementById('p1').innerHTML = datos[0].name + " " + datos[0].descrip;
14        }
15    })
16})
17
18 function getProd(idProd) {
19     const peticion = new XMLHttpRequest();
20     peticion.open('GET', SERVER + '/productos?id=' + idProd);
21     peticion.send();
22     peticion.addEventListener('load', function() {
23         if (peticion.status === 200) {
24             const datos = JSON.parse(peticion.responseText); // Convertirmos los datos JSON a un objeto
25             console.log(datos);
26             return datos
27         } else {
28             console.error("Error " + peticion.status + " (" + peticion.statusText + ") en la petición");
29         }
30     })
31     peticion.addEventListener('error', () => console.error('Error en la petición HTTP'));
32 }

```

En la página se observa que aunque hagamos click varias veces, el contenido párrafo no se actualiza:

Id Producto:

nada

Por otro lado, viendo el terminal, observamos que la primera vez que hacemos click con el id=1, devuelve un código 200 (ha sido correcto el envío) y las siguientes veces devuelve código 304 (la información está en la caché porque es la misma petición y no hubo cambios en esos datos desde que devolvió código 200).

```

Loading .\productos.json
Done

Resources
http://localhost:4000/productos

Home
http://localhost:4000

Type s + enter at any time to create a snapshot of the database
GET /productos?id=2 200 11.331 ms - 65
GET /productos?id=2 304 24.557 ms - -
GET /productos?id=2 304 37.382 ms - -

```

Si miramos en la consola, se aprecia que en la línea 25 los datos obtenidos son los mismos, al *parsear* se obtiene un array de longitud 1, cuyo único elemento es el objeto que buscamos.

```
undefined                                         get200bad.js:11
✖ ► Uncaught TypeError: Cannot read properties of undefined (reading '0')
  at HTMLElement.<anonymous> (get200bad.js:13:54)
▶ [{}]
undefined                                         get200bad.js:11
✖ ► Uncaught TypeError: Cannot read properties of undefined (reading '0')
  at HTMLElement.<anonymous> (get200bad.js:13:54)
▶ [{}]
undefined                                         get200bad.js:11
✖ ► Uncaught TypeError: Cannot read properties of undefined (reading '0')
  at HTMLElement.<anonymous> (get200bad.js:13:54)
                                         get200bad.js:25
▼ [{}]
▶ 0: {name: 'Teclado', descrip: 'Teclado mecánico Cherry ps/2', id: 2}
  length: 1
▶ [[Prototype]]: Array(0)
```

Pero también se observa que la línea 11 recibe *undefined*, y más tarde, aparece el array que vemos en la línea 25. Lo que explica el error de la línea 13: intenta pintar los datos que aún no ha recibido porque la respuesta tarda en llegar. Este error debemos corregirlo.

Visto de otro modo: El array datos es *undefined* en la línea 11 porque cuando se llama a *getProd(idProd)*, esta función no devuelve nada de inmediato, sino que devuelve tiempo después, cuando el servidor contesta, pero entonces nadie está escuchando.

La solución es que todo el código, no sólo de la petición Ajax sino también el de qué hacer con los datos cuando llegan, se encuentre en la función que pide los datos al servidor.

Está resuelto en el apartado siguiente:

## Formulario para obtener (GET) un producto dado su id:

Este ejemplo resuelve el problema del apartado anterior.

```
<form id="addProduct">
  <label for="id-prod">Id Producto: </label><input type="number" name="id-prod" id="id-prod">
  <br>
  <button type="submit">Buscar</button>
  <p id="p1">Aquí vendrán los datos</p>
</form>
```

```
const SERVER = 'http://localhost:4000';

window.addEventListener('load', function() {
  document.getElementById('addProduct').addEventListener('submit', (event) => {
    event.preventDefault(); // Cancela la acción predeterminada del evento 'submit'
    const idProd = document.getElementById('id-prod').value;
    if (isNaN(idProd) || idProd == '') {
      alert('Debes introducir un número');
    } else {
      getProducto(idProd);
    }
  });
});
```

```

        }
    })
})

function getProducto(idProd) {
    const peticion = new XMLHttpRequest();
    peticion.open('GET', SERVER + '/productos?id=' + idProd);
    peticion.send();
    peticion.addEventListener('load', function() {
        if (peticion.status === 200) {
            const datos = JSON.parse(peticion.responseText); // Convertirmos los datos JSON a un objeto
            document.getElementById('p1').innerHTML = datos[0].name+ " "+datos[0].descrip;
            console.log(datos);
        } else {
            console.error("Error " + peticion.status + " (" + peticion.statusText + ") en la petición");
        }
    })
    peticion.addEventListener('error', () => console.error('Error en la petición HTTP'));
}

```

**Nota:** Al poner `event.preventDefault()` en la función asociada a un evento, se cancela la acción predeterminada que tuviera ese evento.

Para pintar los datos, hay que tener en cuenta que GET devuelve un array (en este caso con un único elemento).

Este código funciona correctamente, pero tiene una pega: tenemos que tratar los datos (en este caso pintarlos en el párrafo) en la función que gestiona la petición, porque es la que sabe cuándo están disponibles esos datos. Por tanto, nuestro código es poco claro.

Esto se podría mejorar usando una función **callback**.

## Funciones callback

La idea es que creamos una función que procese los datos (`renderProd`) y se la pasamos a `getProd` como una función **callback** para que la llame cuando tenga los datos:

```

const SERVER = 'http://localhost:4000';

window.addEventListener('load', function() {
    document.getElementById('addProduct').addEventListener('submit', (event) => {
        event.preventDefault();
        let idProd = document.getElementById('id-prod').value;
        if (isNaN(idProd) || idProd.trim() == '') {
            alert('Debes introducir un número')
        } else {
            getProd(idProd, renderProd)
        }
    })
})

function renderProd(datos) {
    // aquí pintamos los datos. Habrá casos que será muy extenso.
    document.getElementById('p1').innerHTML = datos[0].name+ " "+datos[0].descrip;
}

```

```

}

function getProd(idProd, callback) {
  const peticion = new XMLHttpRequest()
  peticion.open('GET', SERVER + '/productos?id=' + idProd);
  peticion.send()
  peticion.addEventListener('load', function() {
    if (peticion.status === 200) {
      callback(JSON.parse(peticion.responseText));
    } else {
      console.error("Error " + peticion.status + " (" + peticion.statusText + ") en la petición")
    }
  })
  peticion.addEventListener('error', () => console.error('Error en la petición HTTP'))
}

```

Hemos creado una función que se ocupa de renderizar (pintar) los datos y se la pasamos a la función que gestiona la petición para que la llame cuando los datos están disponibles (cuando petición.status==200).

Utilizando la función *callback* hemos conseguido que *getProd()* se encargue sólo de obtener los datos y cuando los tenga se los pasa a la función encargada de pintarlos en el documento de la página manipulando el DOM.

## Promesas

Sin embargo hay una forma más limpia de resolver una función asíncrona, y es que el código se parezca al primero que hicimos que no funcionaba, donde la función *getProd()* sólo debía ocuparse de obtener los datos y devolverlos a quien se los pidió:

```

...
let idProd = document.getElementById('id-prod').value;
if (isNaN(idProd) || idProd == '') {
  alert('Debes introducir un número');
} else {
  const datos = getProd(idProd);
  // y aquí usamos los datos recibidos para pintar los datos
}
...

```

La nueva forma es convirtiendo a *getProd()* en una ***promesa***.

Cuando se **realiza una llamada** a una promesa, quien la llama puede usar métodos que NO SE EJECUTARÁN hasta que la promesa se haya resuelto, es decir, hasta que el servidor haya contestado. Estos métodos son:

- **.then(function(datos) { ... })**: se ejecuta si la promesa se ha resuelto satisfactoriamente. Su parámetro es una **función**. Esta función recibirá como parámetro los datos que haya devuelto la promesa (que serán los datos pedidos al servidor). En este caso la promesa se los envía con **resolve(...)**
- **.catch(function(datos) { ... })**: se ejecuta si se ha rechazado la promesa (normalmente porque se ha recibido una respuesta errónea del servidor). Ejecuta una función que recibe como parámetro la información pasada por la promesa al ser rechazada (que será información sobre el error producido). En este caso la promesa se los envía con **reject(...)**

De esta manera el código quedaría:

```

let idProd = document.getElementById('id-prod').value;
if (isNaN(idProd) || idProd == '') {
    alert('Debes introducir un número');
} else {
    getProd(idProd)
        // en el .then() estará el código a ejecutar cuando tengamos los datos
        .then((datos) => {
            document.getElementById('p1').innerHTML = datos[0].name + " " + datos[0].descrip;
        })
        // en el .catch() está el tratamiento de errores
        .catch((error) => console.error(error))
}

```

Para convertir a `getProd()` en una promesa solo hay que “envolver” las instrucciones de la función en una promesa.

```

function getProd(idProd){
    return new Promise((resolve, reject)=>{
        // Aquí el contenido de GetProd()
    })
}

```

Esto hace que devuelva un objeto de tipo `Promise` (`return new Promise()`) cuyo único parámetro es una función que recibe 2 parámetros:

- **`resolve`**: función `callback` a la que se llamará cuando se resuelva la promesa satisfactoriamente.
- **`reject`**: función `callback` a la que se llamará si se resuelve la promesa con errores.

El funcionamiento es:

- cuando la promesa se resuelva satisfactoriamente `getProd` llama a la función `resolve()` y le pasa los datos recibidos por el servidor. Esto hace que se ejecute el método `.then` de la llamada a la promesa que recibirá como parámetro esos datos. Se ejecuta `.then()` después de `resolve()`
- si se produce algún error se rechaza la promesa llamando a la función `reject()` y pasando como parámetro la información del fallo producido y esto hará que se ejecute el `.catch` en la función que llamó a la promesa. Se ejecuta `.catch()` después de `resolve()`.

Por tanto, nuestra función `getProd` ahora quedará así:

```

function getPosts(idProd) {
    return new Promise((resolve, reject) => {
        const peticion = new XMLHttpRequest();
        peticion.open('GET', SERVER + '/productos?id=' + idProd);
        peticion.send();
        peticion.addEventListener('load', () => {
            if (peticion.status === 200) {
                resolve(JSON.parse(peticion.responseText));
            } else {
                reject("Error " + peticion.status + " (" + peticion.statusText + ") en la petición");
            }
        })
        peticion.addEventListener('error', () => reject('Error en la petición HTTP'));
    })
}

```

```
}
```

Se observa que el único cambio es la primera línea donde se convierte la función `getProd()` en una **promesa**, y que luego para “devolver” los datos a quien llama a `getProd` en lugar de hacer un `return`, que ya se ha visto que no funciona, se hace un `resolve` si todo ha ido bien o un `reject` si ha fallado.

Desde donde llamamos a la promesa **nos suscribimos a ella** usando los métodos `.then()` y `.catch()` vistos anteriormente.

Básicamente, lo que nos va a proporcionar el uso de promesas es un código más claro y mantenible, ya que el código a ejecutar cuando se obtengan los datos asíncronamente estará donde se piden esos datos y no en una función escuchadora o en una función `callback`.

Utilizando promesas vamos a conseguir que la función que pide los datos sea quien los obtiene y los trate o quien informa si hay un error.

El código del ejemplo de obtener un producto desde su id usando promesas sería el siguiente:

El código HTML sería igual que antes:

```
<form id="addProduct">
    <label for="id-prod">Id Producto: </label><input type="number" name="id-prod" id="id-prod">
    <br>
    <button type="submit">Buscar</button>
    <p id="p1">Aquí vendrán los datos</p>
</form>
```

Y el código Javascript sería:

```
const SERVER = 'http://localhost:4000';

window.addEventListener('load', function() {
    document.getElementById('addProduct').addEventListener('submit', (event) => {
        event.preventDefault();
        let idProd = document.getElementById('id-prod').value
        if (isNaN(idProd) || idProd.trim() == '') {
            alert('Debes introducir un número')
        } else {
            getProd(idProd)
                .then(function(datos) {
                    // aquí pintamos los datos. Habrá casos que será muy extenso.
                    document.getElementById('p1').innerHTML = datos[0].name+ " " +datos[0].descrip;
                })
                .catch(function(error) {
                    console.error(error)
                })
        }
    })
})

function getProd(idProd) {
    return new Promise(function(resolve, reject) {
        let peticion = new XMLHttpRequest()
```

```

peticion.open('GET', SERVER + '/productos?id=' + idProd);
peticion.send()
peticion.addEventListener('load', () => {
  if (peticion.status === 200) {
    resolve(JSON.parse(peticion.responseText))
  } else {
    reject("Error " + this.status + " (" + this.statusText + ") en la petición")
  }
})
peticion.addEventListener('error', () => reject('Error en la petición HTTP'))
})
}

```

**Nota:** Los errores del servidor SIEMPRE llegan a la consola. En el ejemplo anterior si se produce un error de servidor aparecerá 2 veces en la consola: la primera que es el error original y la segunda donde está pintado con `console.error()`.

En este ejemplo, aunque pidamos un id de producto no existente en el servidor, devuelve un código 200 y un array vacío. El error que se ve en consola es porque no puede pintar un array vacío. Este error es capturado por `catch()`, y permite que la página siga funcionando.

Más sobre Promesas en [MDN web docs](#) y en los documentos anexos.

## Fetch

La [API Fetch](#) es una interfaz para manipular la información del servidor en forma de promesas.

Como el código que hay que escribir para hacer una petición Ajax es largo y repetitivo, la API-Fetch permite realizar una petición Ajax genérica que directamente devuelve en forma de **promesa**.

Básicamente lo que hace es encapsular en una función todo el código que se repite siempre en una petición AJAX (crear la petición, hacer el `open`, el `send`, escuchar los eventos, ...).

La función `fetch` es similar a la función `getProd` que hemos creado antes, pero es genérica para que sirva para cualquier petición pasándole la URL. Su código es algo similar a:

```

function fetch(url) {
  return new Promise((resolve, reject) => {
    const peticion = new XMLHttpRequest();
    peticion.open('GET', url);
    peticion.send();
    peticion.addEventListener('load', () => {
      resolve(peticion.responseText);
    })
    peticion.addEventListener('error', () => reject('Network Error'));
  })
}

```

Hay 2 cosas que cambian respecto a nuestra función `getProd()`:

1. `fetch` devuelve los datos “en crudo” por lo que si la respuesta está en formato JSON habrá que convertirlos. Para ello dispone del método `.json()` que funciona como el `JSON.parse()`. Este método devuelve una nueva promesa a la que nos suscribimos con un nuevo `.then`. Es lo que se llama promesas encadenadas. Ejemplo.:

```

fetch('http://localhost:4000/productos?id=' + idProd)
  .then(response => response.json()) // los datos son una cadena JSON
  .then(myData => { // ya tenemos los datos en _myData_ como un objeto o array
    // Aquí procesamos los datos (en nuestro ejemplo los pintaríamos en el párrafo de la página)
    console.log(myData)
  })
  .catch(err => console.error(err));

```

2. *fetch* llama a *resolve* siempre que el servidor conteste, sin comprobar si la respuesta es de éxito (200, 201, 304,...) o de error (4xx, 5xx). Por tanto, siempre se ejecutará el *then* excepto si se trata de un error de red y el servidor no responde (no ha habido respuesta).

### Propiedades y métodos de la respuesta

La respuesta devuelta por *fetch()* tiene las siguientes propiedades y métodos:

- **status**: el código de estado devuelto por el servidor (200, 404, ...)
- **statusText**: el texto correspondiente a ese código (Ok, Not found, ...)
- **ok**: booleano que vale *true* si el status está entre 200 y 299 y *false* en caso contrario
- **json()**: devuelve una promesa que se resolverá con los datos de la respuesta convertidos a un objeto (a los datos les realiza un *JSON.parse()*)
- otros métodos para convertir los datos según el formato que tengan: **text()**, **blob()**, **formData()**, ... Todos devuelven una promesa con los datos de distintos formatos convertidos.

El ejemplo que hemos visto con las promesas, usando *fetch* quedaría:

El código HTML sería igual que antes:

```

<form id="addProduct">
  <label for="id-prod">Id Producto: </label><input type="number" name="id-prod" id="id-prod">
  <br>
  <button type="submit">Buscar</button>
  <p id="p1">Aquí vendrán los datos</p>
</form>

```

Y el código Javascript sería:

```

const SERVER = 'http://localhost:4000';

window.addEventListener('load', () => {
  document.getElementById('addProduct').addEventListener('submit', (event) => {
    event.preventDefault();
    let idProd = document.getElementById('id-prod').value;
    if (isNaN(idProd) || idProd.trim() == '') {
      alert('Debes introducir un número')
    } else {
      fetch(SERVER + '/productos?id=' + idProd)
        .then((response) => response.json())
        .then((datos) => {
          // aquí pintamos los datos. Habrá casos que será muy extenso.
          document.getElementById('p1').innerHTML = datos[0].name + " " + datos[0].descrip;
        })
    }
  })
})

```

```
    })
    .catch((error) => console.error(error))
  }
})
})
```

Pero este ejemplo fallaría si hubiéramos puesto mal la *url* ya que contestaría con un 404 y fallaría al ejecutar el *then* intentando pintar un producto que no tenemos.

## *Gestión de errores con fetch*

Según [MDN](#), la promesa devuelta por la API `fetch` sólo es rechazada en el caso de un error de red, es decir, el `.catch` sólo saltará si no hemos recibido respuesta del servidor; en caso contrario la promesa siempre es resuelta.

Por tanto, para saber si se ha resuelto satisfactoriamente o no, debemos comprobar la propiedad `.ok` de la respuesta. El código correcto del ejemplo anterior gestionando los posibles errores del servidor sería:

```
// tratando los errores en fetch
const SERVER = 'http://localhost:4000';

window.addEventListener("load", () => {
  document.getElementById("addProduct").addEventListener("submit", (event) => {
    event.preventDefault();
    let idProd = document.getElementById("id-prod").value;
    if (isNaN(idProd) || idProd.trim() == "") {
      alert("Debes introducir un número");
    } else {
      fetch(SERVER + "/productos?id=" + idProd)
        .then((response) => {
          if (!response.ok) {
            // lanzamos un error que interceptará el .catch()
            throw `Error ${response.status} de la BBDD: ${response.statusText}`;
          }
          return response.json(); // devolvemos la promesa que hará el JSON.parse
        })
        .then((datos) => {
          // ya tenemos los datos formateados
          // Aquí procesamos los datos (en nuestro ejemplo los pintaríamos en la página)
          document.getElementById("p1").innerHTML =
            datos[0].name + " " + datos[0].descrip;
          console.log(datos);
        })
        .catch((error) => console.error(error));
    }
  });
});
```

En este caso, si la respuesta del servidor no es `ok` lanzamos un error que es interceptado por nuestro propio `catch`

### Otros métodos de petición POST, PUT...

Los ejemplos anteriores hacen peticiones GET al servidor. Para peticiones que no sean GET la función `fetch()` admite un segundo parámetro: un objeto con la información a enviar en la petición HTTP. Ej.:

```
// Otros métodos de petición POST, PUT...
fetch(url, {
  method: 'POST', // o 'PUT', 'GET', 'DELETE'
  body: JSON.stringify(data), // los datos que enviamos al servidor en el 'send'
  headers:{
    'Content-Type': 'application/json'
  }
}).then
```

Ejemplo de una petición para añadir datos:

```
fetch(url, {
  method: 'POST',
  body: JSON.stringify(data), // los datos que enviamos al servidor en el 'send'
  headers:{
    'Content-Type': 'application/json'
  }
})
.then(response => {
  if (!response.ok) {
    throw `Error ${response.status} de la BBDD: ${response.statusText}`
  }
  return response.json()
})
.then(datos => {
  alert('Datos recibidos')
  console.log(datos)
})
.catch(err => {
  alert('Error en la petición HTTP: '+err.message);
})
```

Más ejemplos en [MDN web docs](#)

Realiza una página web para que cualquier usuario de la página pueda dar de alta nuevos productos en `productos.json` utilizando `fetch`. El usuario introducirá el nombre y la descripción del producto, la API asignará el id que corresponda.

En la solución del ejercicio, si añadimos al objeto producto que se va a enviar con POST la propiedad `id` con un valor puesto a mano, puede ocurrir:

- Si el número de `id` se repite, provocará un error 500, y el producto no se añadirá a la BBDD.
- Si el `id` no existe, se creará el producto con el valor `id` que se haya puesto a mano.
- Si no se pone `id`, la API colocará el número siguiente al último `id` del fichero. (Cuidado!!)



## DWEC - Javascript Web Cliente.

JavaScript - Ajax 4 .....	1
async / await.....	1
Gestión de errores en async/await .....	3
Ejemplos de GET y POST con Fetch (utilizando y sin usar async-await) .....	4
Con nuestro fichero productos.json:.....	4
Index.html para GET.....	4
Código javascript.....	5
Index.html para POST.....	6
Código JavaScript .....	6
Hacer varias peticiones simultáneamente. Promise.all .....	8
Single Page Application .....	9
Resumen de llamadas asíncronas .....	10
CORS .....	10

# JavaScript - Ajax 4

## async / await

Son instrucciones nuevas introducidas en ES2017 que permiten escribir el código de peticiones asíncronas como si fueran síncronas, lo que facilita su comprensión.

Hay que tener en cuenta que NO están soportadas por navegadores antiguos.

Si hubiéramos utilizado `async/await` en el primer ejemplo que hicimos, sí habría funcionado.

Se puede llamar a cualquier función asíncrona (por ejemplo, una promesa como `fetch`) anteponiendo la palabra **await** a la llamada. Esto provocará que la ejecución se “espere” a que se resuelva la promesa devuelta por esa función. Así nuestro código se asemeja a código síncrono ya que no continúan ejecutándose las instrucciones que hay después de un `await` hasta que esa petición se ha resuelto.

Cualquier función que realice un `await` pasa a ser asíncrona ya que no se ejecuta en ese momento, sino que se espera un tiempo.

Para indicarlo debemos anteponer la palabra **async** a la declaración `function`. Al hacerlo, automáticamente se “envuelve” esa función en una promesa (o sea, que esa función pasa a devolver una promesa, a la que podríamos ponerle un `await` o un `.then()`).

Siguiendo con el ejemplo anterior, el código Javascript sería:

```
async function pideDatos() {
    const response = await fetch('http://localhost:4000/productos?id=' + idProd);
    if (!response.ok) {
```

```

        throw `Error ${response.status} de la BBDD: ${response.statusText}`
    }
    const myData = await response.json(); // recuerda que .json() tb es una promesa
    return myData;
}
...
// Y llamaremos a esa función con
const myData = await pideDatos();

```

**Nota:** solo se puede utilizar **await** dentro de funciones declaradas con la palabra reservada **async**.

Observa la diferencia: si hacemos:

```
const response = fetch('http://localhost:4000/productos?id=' + idProd);
```

obtenemos en *response* una promesa, y para obtener el valor se debería hacer *response.then()*. Pero si hacemos:

```
const response = await fetch('http://localhost:4000/productos?id=' + idProd);
```

lo que obtenemos en *response* es ya el valor devuelto por la promesa cuando se resuelve.

Con esto conseguimos que llamadas asíncronas se comporten como instrucciones síncronas, lo que aporta claridad al código.

Hay algunos ejemplos del uso de *async / await* en la [página de MDN](#).

Siguiendo con el ejemplo de obtener datos de un producto indicando su id:

El código HTML sería igual que antes:

```

<form id="getProduct">
    <label for="id-prod">Id Producto: </label><input type="number" name="id-prod" id="id-prod">
    <br>
    <button type="submit">Buscar</button>
    <p id="p1">Aquí vendrán los datos</p>
</form>

```

Y el código Javascript sería:

```

const SERVER = 'http://localhost:4000';

window.addEventListener('load', () => {
    document.getElementById('getProduct').addEventListener('submit', async (event) => {
        event.preventDefault();
        let idProd = document.getElementById('id-prod').value
        if (isNaN(idProd) || idProd.trim() == '') {
            alert('Debes introducir un número')
        } else {
            const datos = await getData(idProd)
            // La ejecución se para en la sentencia anterior hasta que
            // contesta la función getData
            document.getElementById('p1').innerHTML = datos[0].name+ " " +datos[0].descrip;
        }
    })
}

```

```

    })
}

async function getData(idProd) {
  const response = await fetch(SERVER + '/productos?id=' + idProd)
  if (!response.ok) {
    throw `Error ${response.status} de la BBDD: ${response.statusText}`
  }
  const datos = await response.json()
  return datos
}

```

### Gestión de errores en async/await

En el código anterior no estamos tratando los posibles errores que se pueden producir.

Con *async / await* los errores se tratan como en las excepciones, con *try ... catch*.

El código que resulta para el ejemplo de pintar los datos de un producto indicando su id:

El código HTML sería igual que siempre:

```

<form id="getProduct">
  <label for="id-prod">Id Producto: </label><input type="number" name="id-prod" id="id-prod">
  <br>
  <button type="submit">Buscar</button>
  <p id="p1">Aquí vendrán los datos</p>
</form>

```

Y el código Javascript sería:

```

// fetch con async/await
// tratando errores con try-catch
const SERVER = 'http://localhost:4000'

window.addEventListener('load', () => {
  document.getElementById('getProduct').addEventListener('submit', async (event) => {
    event.preventDefault();
    let idProd = document.getElementById('id-prod').value
    if (isNaN(idProd) || idProd.trim() == '') {
      alert('Debes introducir un número')
    } else {
      try {
        const datos = await getData(idProd)
        // La ejecución se para en la sentencia anterior hasta que
        // contesta la función getData
        document.getElementById('p1').innerHTML = datos.name + " " + datos.descrip;
      } catch (err) {
        console.log("mal");
        console.error(err);
        return;
      }
    }
  })
})

```

```
async function getData(idProd) {
  const response = await fetch(SERVER + '/productos?id=' + idProd)
  if (!response.ok) {
    throw `Error ${response.status} de la BBDD: ${response.statusText}`
  }
  const datos = await response.json()
  return datos
}
```

También podemos tratar los errores sin usar `try...catch`, porque como una función asíncrona devuelve una promesa, podemos suscribirnos directamente a su `.catch`

## Ejemplos de GET y POST con Fetch (utilizando y sin usar async-await)

*Con nuestro fichero productos.json:*

```
{
  "productos": [
    {
      "id": 1,
      "name": "Teclado",
      "descrip": "Teclado mecánico Cherry ps/2"
    },
    {
      "id": 2,
      "name": "Monitor Ph-21",
      "descrip": "Monitor Phillips SVGA 21 \""
    },
    {
      "id": 3,
      "name": "Ratón Logi-Laser",
      "descrip": "Tatón Láser Logitech Pro USB"
    }
  ]
}
```

*Index.html para GET*

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
<body>
  <form id="getProduct">
    <label for="id-prod">Id Producto: </label><input type="number" name="id-prod" id="id-prod">
    <br>
```

```

        <button type="submit">Buscar</button>
        <p id="p1">Aquí vendrán los datos</p>
    </form>
    <script src="js/get.js"></script>

</body>
</html>

```

### Código javascript

<pre> // GET con fetch const SERVER= 'http://127.0.0.1:4000'; window.addEventListener("load", ()=&gt;{     document.getElementById('getProduct')         .addEventListener('submit', (event)=&gt;{             event.preventDefault();             let idProd=document.getElementById('id-prod').value;             let promesa=fetch(SERVER + '/productos?id=' + idProd);                 //obtenemos una promesa             promesa                 .then((dato1)=&gt;dato1.json())                     //ejecuta lo de dentro, dato1 es lo que then ha                     //recibido                     .then((dato2)=&gt;{ // dato2 es lo que recibe del                         anterior then                         console.log(dato2);                         document.getElementById('p1').innerText=dato2[0].nam e;                     })                     .catch((problema)=&gt; document.getElementById('p1')                         .innerText="Ha habido error: "+problema);             });         });     }); </pre>	<pre> // fetch con async await // se trata de separar la zona donde // se pinta, de la petición a la API. // ya no usamos then, porque await garantiza // que ya llegó la respuesta const SERVER= 'http://127.0.0.1:4000'; window.addEventListener("load", ()=&gt;{     document.getElementById('getProduct')         .addEventListener('submit', async (event)=&gt;{             event.preventDefault();             const idProd=document.getElementById('id- prod').value;             const dato1=await getData(idProd);             document.getElementById('p1').innerText=dato1[ 0]                 .name + " " +dato1[0].descrip;         });     });  async function getData(idProd){     const dato1 = await fetch(SERVER+'/productos?id='+idProd);     // lo normal sería mirar si hubo error:     /* if (!dato1.ok) {         throw `error\${dato1.statusText}`     }*/     console.log(dato1);     const dato2 = await dato1.json();     console.log(dato2);     return dato2; } </pre>
--	--

```
> JS get.js > ...
1 // fetch con
2 const SERVER= 'http://127.0.0.1:4000';
3 window.addEventListener("load", ()=>{
4   document.getElementById('getProduct').addEventListener('submit', (event)=>{
5     event.preventDefault();
6     let idProd=document.getElementById('id-prod').value;
7     let promesa=fetch(SERVER + '/productos?id=' + idProd); //obtenemos una promesa
8     promesa
9       .then((dato1)=>dato1.json()) //ejecuta lo de dentro, dato1 es lo que then ha recibido
10      .then((dato2)=>{ // dato2 es lo que recibe del anterior then
11        console.log(dato2);
12        document.getElementById('p1').innerText=dato2[0].name;
13      })
14      .catch((problema)=> document.getElementById('p1').innerText="Ha habido error: "+problema);
15    });
16  });
17});
```

```
js > JS get.js > getData
1 // fetch con async await
2 // se trata de separar la zona donde se pinta, de la petición a la API.
3 // ya no usamos then, porque await garantiza que ya llegó la respuesta
4 const SERVER= 'http://127.0.0.1:4000';
5 window.addEventListener("load", ()=>{
6   document.getElementById('getProduct').addEventListener('submit', async (event)=>{
7     event.preventDefault();
8     const idProd=document.getElementById('id-prod').value;
9     const dato1=await getData(idProd);
10    document.getElementById('p1').innerText=dato1[0].name + " " +dato1[0].descrip;
11  });
12 });
13
14 async function getData(idProd){
15   const dato1 = await fetch(SERVER + '/productos?id=' + idProd);
16   // lo normal sería mirar si hubo error:
17   // if (!dato1.ok) { throw `error${dato1.status} ${dato1.statusText}`}
18   console.log(dato1);
19   const dato2 = await dato1.json();
20   console.log(dato2);
21   return dato2;
22 }
```

### Index.html para POST

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <link rel="shortcut icon" href="#" type="image/x-icon">
  <title>Document</title>
</head>
<body>
  <form id="addProduct">
    <label for="name">Nombre: </label><input type="text" name="name" id="name" required><br>
    <label for="descrip">Descripción: </label><input type="text" name="descrip" id="descrip" required><br>

    <button type="submit">Agregar Producto</button>
    <p id="p1"> ...</p>
  </form>
  <script src="js/post.js"></script>
</body>
</html>
```

### Código JavaScript

// POST con fetch	// POST con fetch y asunc await
<pre>const SERVER = 'http://localhost:4000';  window.addEventListener('load', ()=&gt;{   document.getElementById('addProduct').addEventListener('submit', (event)=&gt;{     event.preventDefault();     const nuevoProducto={       id:"",       name:document.getElementById('name').value,       descrip:document.getElementById('descrip').value     };   }); });</pre>	<pre>const SERVER = 'http://localhost:4000';  async function anadirProducto(nuevoProducto){   const dato1 = await fetch(SERVER+ '/productos',   {     method: 'POST',     body: JSON.stringify(nuevoProducto),     headers: {       'Content-Type': 'application/json'     }   });   const dato2= await dato1.json();   return dato2; }</pre>

```

const promesa=fetch(SERVER+'/productos', {
  method: 'POST',
  body: JSON.stringify(nuevoProducto),
  headers: {
    'Content-Type': 'application/json'
  }
})
promesa
  .then((dato1)=>dato1.json())
  .then((dato2)=>{
    //pintamos lo que queramos
    console.log(dato2);
    document.getElementById('p1').innerText=`${dato2.name}
${dato2.descrip}`;
  })
  .catch();
});
});
```

```

}

window.addEventListener('load', ()=>{
  document.getElementById('addProduct')
    .addEventListener('submit', async (event)=>{
      event.preventDefault();
      const nuevoProducto={
        id:"",
        name:document.getElementById('name').value,
        descrip:document.getElementById('descrip').value
      };
      const dato2 = await anadirProducto(nuevoProducto);

      //pintamos lo que queramos
      console.log(dato2);
      document.getElementById('p1')
        .innerText=`${dato2.name}
${dato2.descrip}`;
    });
});
```

```

> JS post.js > ...
1 // POST con fetch
2 const SERVER = 'http://localhost:4000';
3 window.addEventListener('load', ()=>{
4   document.getElementById('addProduct')
5     .addEventListener('submit', (event)=>{
6     event.preventDefault();
7     const nuevoProducto={
8       id:"",
9       name:document.getElementById('name').value,
10      descrip:document.getElementById('descrip').value
11    };
12
13    const promesa=fetch(SERVER+'/productos', {
14      method: 'POST',
15      body: JSON.stringify(nuevoProducto),
16      headers: {
17        'Content-Type': 'application/json'
18      }
19    })
20    promesa
21      .then((dato1)=>dato1.json())
22      .then((dato2)=>{
23        //pintamos lo que queramos
24        console.log(dato2);
25        document.getElementById('p1')
26          .innerText=`${dato2.name} ${dato2.descrip}`;
27      })
28      .catch();
29  });
30});
```

```

: > JS post.js > ...
1 // POST con fetch y asinc await
2 const SERVER = 'http://localhost:4000';
3 async function anadirProducto(nuevoProducto){
4   const dato1 = await fetch(SERVER+'/productos', {
5     method: 'POST',
6     body: JSON.stringify(nuevoProducto),
7     headers: {
8       'Content-Type': 'application/json'
9     }
10   );
11   const dato2= await dato1.json();
12   return dato2;
13 }
14
15 window.addEventListener('load', ()=>{
16   document.getElementById('addProduct')
17     .addEventListener('submit', async (event)=>{
18     event.preventDefault();
19     const nuevoProducto={
20       id:"",
21       name:document.getElementById('name').value,
22       descrip:document.getElementById('descrip').value
23     };
24
25     const dato2 = await anadirProducto(nuevoProducto);

26     //pintamos lo que queramos
27     console.log(dato2);
28     document.getElementById('p1')
29       .innerText=`${dato2.name} ${dato2.descrip}`;
30   });
31 });
32 }
```

## Hacer varias peticiones simultáneamente. Promise.all

En ocasiones necesitamos hacer más de una petición al servidor.

Por ejemplo; para obtener los productos y sus categorías podríamos hacer:

```
function getTable(table) {
    return new Promise((resolve, reject) => {
        fetch(SERVER + table)
            .then(response => {
                if (!response.ok) {
                    throw `Error ${response.status} de la BBDD: ${response.statusText}`
                }
                return response.json()
            })
            .then((data) => resolve(data))
            .catch((error) => reject(error))
    })
}

function getData() {
    getTable('/categories')
        .then((categories) => categories.forEach((category) => renderCategory(category)))
        .catch((error) => renderErrorMessage(error))
    getTable('/products')
        .then((products) => products.forEach((product) => renderProduct(product)))
        .catch((error) => renderErrorMessage(error))
}
```

Pero si para renderizar los productos necesitamos tener las categorías, este código no nos lo garantiza ya que el servidor podría devolver antes los productos, aunque los hemos pedido después.

Una solución sería no pedir los productos hasta tener las categorías:

```
function getData() {
    getTable('/categories')
        .then((categories) => {
            categories.forEach((category) => renderCategory(category))
            getTable('/products')
                .then((products) => products.forEach((product) => renderProduct(product)))
                .catch((error) => renderErrorMessage(error))
        })
        .catch((error) => renderErrorMessage(error))
}
```

pero esto hará más lento nuestro código al no hacer las 2 peticiones simultáneamente.

La solución es usar el método `Promise.all()` al que se le pasa un array de promesas a hacer y devuelve una promesa que:

- se resuelve en el momento en que todas las promesas se han resuelto satisfactoriamente o
- se rechaza en el momento en que alguna de las promesas es rechazada

El código anterior de forma correcta sería:

```
function getData() {
  Promise.all([
    getTable('/categories')
    getTable('/products')
  ])
  .then(([categories, products]) => {
    categories.forEach((category) => renderCategory(category))
    products.forEach((product) => renderProduct(product))
  })
  .catch((error) => renderErrorMessage(error))
}
```

Lo mismo pasa si en vez de promesas usamos *async/await*. Si hacemos:

```
async function getTable(table) {
  const response = await fetch(SERVER + table)
  if (!response.ok) {
    throw `Error ${response.status} de la BBDD: ${response.statusText}`
  }
  const data = await response.json()
  return data
}

async function getData() {
  const responseCategories = await getTable('/categories');
  const responseProducts = await getTable('/products');
  categories.forEach((category) => renderCategory(category))
  products.forEach((product) => renderProduct(product))
}
```

tenemos el problema de que no comienza la petición de los productos hasta que se reciben las categorías. La solución con `Promise.all()` sería:

```
async function getData() {
  const [categories, products] = await Promise.all([
    getTable('/categories')
    getTable('/products')
  ])
  categories.forEach((category) => renderCategory(category))
  products.forEach((product) => renderProduct(product))
}
```

## Single Page Application

Ajax es la base para construir SPAs que permiten al usuario interactuar con una aplicación web como si se tratara de una aplicación de escritorio (sin “esperas” que dejen la página en blanco o no funcional mientras se recarga desde el servidor).

En una SPA sólo se carga la página de inicio (es la única página que existe) que se va modificando y cambiando sus datos como respuesta a la interacción del usuario.

Para obtener los nuevos datos se realizan peticiones al servidor (normalmente Ajax). La respuesta son datos (JSON, XML, ...) que se muestran al usuario modificando mediante DOM la página mostrada (o podrían ser trozos de HTML que se cargan en determinadas partes de la página, o ...).

## Resumen de llamadas asíncronas

Una llamada Ajax es un tipo de llamada asíncrona que podemos hacer en Javascript.

Aunque hay otros tipos de llamadas asíncronas, como un `setTimeout()` o las funciones manejadoras de eventos.

Como hemos visto, para la gestión de las llamadas asíncronas tenemos varios métodos y los más comunes son:

- funciones *callback*
- *promesas*
- *async / await*

Cuando se produce una llamada asíncrona el orden de ejecución del código no es el que vemos en el programa, ya que el código de respuesta de la llamada no se ejecutará hasta completarse ésta.

Además, si hacemos varias llamadas tampoco sabemos en qué orden se ejecutarán sus respuestas, ya que depende de cuándo finalice cada una.

Si usamos funciones *callback* y necesitamos que cada función no se ejecute hasta que haya terminado la anterior, debemos llamarla en la respuesta a la función anterior, lo que provoca un tipo de código difícil de leer llamado “*callback hell*”.

Para evitar estos problemas surgieron las ***promesas*** que permiten evitar las funciones *callback* tan difíciles de leer. Y si necesitamos ejecutar secuencialmente las funciones evitaremos la pirámide de llamadas *callback*.

Aun así, el código no es muy claro. Para mejorarlo tenemos ***async y await***. Estas funciones forman parte del estándar ES2017 por lo que no están soportadas por navegadores muy antiguos (aunque siempre podemos transpilar con *Babel*).

## CORS

*Cross-Origin Resource Sharing (CORS)* es un mecanismo de seguridad que incluyen los navegadores y que por defecto impiden que se puedan realizar peticiones Ajax desde un navegador a un servidor con un dominio diferente al de la página cargada originalmente.

Si necesitamos hacer este tipo de peticiones necesitamos que el servidor al que hacemos la petición añada en su respuesta la cabecera *Access-Control-Allow-Origin* donde indiquemos el dominio desde el que se pueden hacer peticiones (o \* para permitirlas desde cualquier dominio).

El navegador comprobará las cabeceras de respuesta y si el dominio indicado por ella coincide con el dominio desde el que se hizo la petición, ésta se permitirá.

Como en desarrollo normalmente no estamos en el dominio de producción (para el que se permitirán las peticiones) podemos instalar en el navegador la extensión *allow CORS* que al activarla deshabilita la seguridad CORS en el navegador.

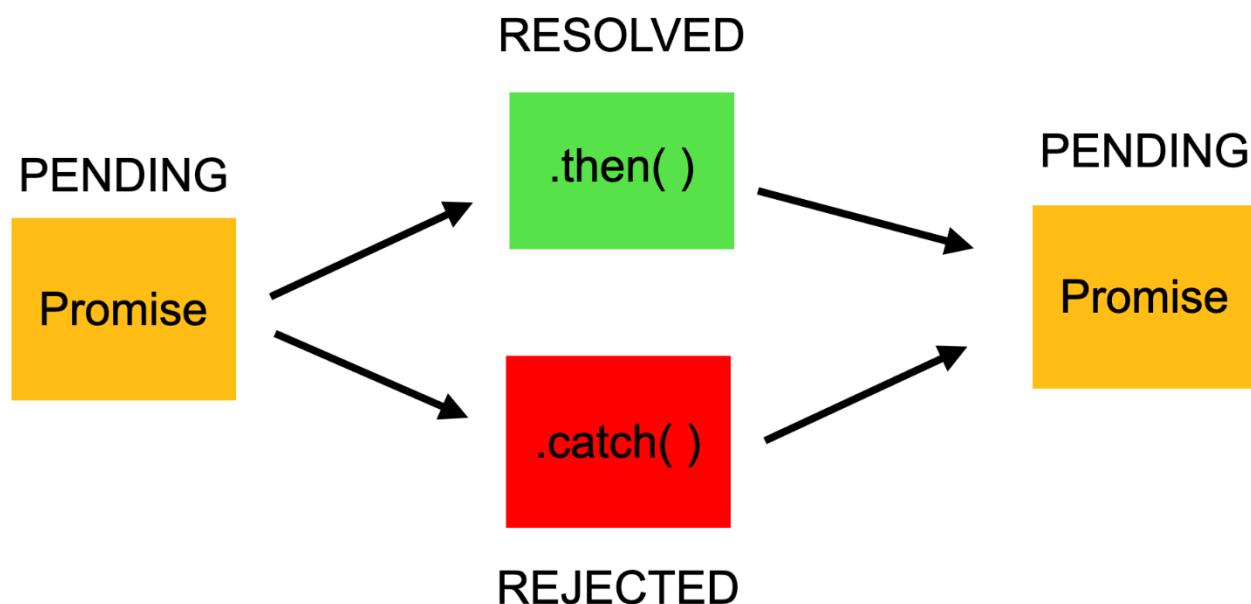
## JavaScript - Promesas

# Promesas en JavaScript

Una Promesa es un objeto. Hay 3 estados del objeto Promesa:

- Pendiente: estado inicial, antes de que la promesa tenga éxito o falle
- Resuelto: promesa completada
- Rechazado: promesa fallida

Veamos una representación de los procesos de Promesas:



Cuando solicitamos datos del servidor mediante una Promesa, estará en modo pendiente hasta que recibamos nuestros datos.

Si logramos obtener la información del servidor, la Promesa se resolverá con éxito. Pero si no obtenemos la información, entonces la Promesa estará en el estado rechazado.

Además, si hay múltiples solicitudes, luego de que se resuelva (o rechace la primera Promesa), comenzará un nuevo proceso al que podemos adjuntarla directamente mediante un método llamado **encadenamiento**.

La principal **diferencia** entre las funciones de **retrollamadas** y las **promesas** es que adjuntamos una retrollamada a una promesa en lugar de pasarlala. Así que todavía usamos funciones de retrollamada (Callback) con Promesas, pero de una manera diferente (encadenamiento).

## Encadenamiento:

Las funciones Callback se han utilizado solas para operaciones asincrónicas en JavaScript durante muchos años. Pero en algunos casos, usar Promesas puede ser una mejor opción.

Si hay varias operaciones asincrónicas por hacer y si tratamos de usar retrollamadas antiguas para ellas, nos encontraremos rápidamente dentro de una situación llamada "Infierno Retrollamada":

```
firstRequest(function(response) {
    secondRequest(response, function(nextResponse) {
        thirdRequest(nextResponse, function(finalResponse) {
            console.log('Final response: ' + finalResponse);
        }, failureCallback);
    }, failureCallback);
}, failureCallback);
```

Sin embargo, si manejamos la misma operación con Promesas, ya que podemos adjuntar retrollamadas en lugar de pasárlas, esta vez el mismo código anterior parece mucho más limpio y fácil de leer:

```
firstRequest()
  .then(function(response) {
    return secondRequest(response);
}).then(function(nextResponse) {
  return thirdRequest(nextResponse);
}).then(function(finalResponse) {
  console.log('Final response: ' + finalResponse);
}).catch(failureCallback);
```

El código anterior muestra cómo se pueden encadenar múltiples retrollamadas una tras otra. El encadenamiento es una de las mejores características de Promesas.

## Creación y uso de una promesa paso a paso

En primer lugar, usamos un constructor para crear un objeto Promesa:

```
const myPromise = new Promise();
```

Se necesitan dos parámetros, uno para el éxito (resolver) y otro para el error (rechazar):

```
const myPromise = new Promise((resolve, reject) => {
  // condition
```

```
});
```

Finalmente, habrá una condición. Si se cumple la condición, la Promesa se resolverá, de lo contrario será rechazada:

```
const myPromise = new Promise((resolve, reject) => {
  let condition;

  if(condition is met) {
    resolve('Promise is resolved successfully.');
  } else {
    reject('Promise is rejected');
  }
});
```

Ya está creada una promesa, vamos a usarla:

## then( ) para promesas resueltas:

En la imagen del principio se observa que hay 2 casos: uno para promesas resueltas y otro para rechazadas. Si la Promesa se resuelve (caso de éxito), algo sucederá después (depende de lo que hagamos con la Promesa exitosa).

```
myPromise.then();
```

Se llama al método `then()` después de que se resuelva la Promesa. Entonces podemos decidir qué hacer con la Promesa resuelta.

Por ejemplo, registremos el mensaje en la consola lo que obtuvimos de la Promesa:

```
myPromise.then((message) => {
  console.log(message);
});
```

## catch( ) para Promesas rechazadas:

Sin embargo, el método `then()` es solo para Promesas resueltas. ¿Qué pasa si la Promesa falla? Entonces, necesitamos usar el método `catch()`.

Del mismo modo que adjuntamos el método `then()`. También podemos adjuntar directamente el método `catch()` justo después de `then()`:

```
myPromise.then((message) => {
```

```
    console.log(message);
}).catch((message) => {
    console.log(message);
});
```

Entonces si la promesa es rechazada, saltará al método catch() y esta vez veremos un mensaje diferente en la consola.

Traducido y adaptado del artículo de Cem Eygi - JavaScript Promise Tutorial: Resolve, Reject, and Chaining in JS and ES6.



## DWEC - Javascript Web Cliente.

# MVC - El patrón Modelo-Vista-Controlador

MVC - El patrón Modelo-Vista-Controlador.....	1
Una aplicación sin MVC.....	1
Nuestro patrón MVC.....	2
main.js.....	3
model/store.class.js .....	3
model/product.class.js.....	3
view/index.js .....	4
controller/index.js.....	4

Modelo-vista-controlador (MVC) es el patrón de arquitectura de software más utilizado en la actualidad en desarrollo web (y también en muchas aplicaciones de escritorio). Este patrón propone separar la aplicación en tres componentes distintos: el modelo, la vista y el controlador:

- El modelo es el conjunto de todos los datos o información con la que trabaja la aplicación. Normalmente serán variables extraídas de una base de datos y el modelo gestiona los accesos a dicha información, tanto consultas como actualizaciones, implementando también los privilegios de acceso que se hayan descrito en las especificaciones de la aplicación (lógica de negocio). Normalmente el modelo no tiene conocimiento de las otras partes de la aplicación.
- La vista muestra al usuario el modelo (información y lógica de negocio) en un formato adecuado para interactuar (usualmente la interfaz de usuario). Es la intermedia entre la aplicación y el usuario
- El controlador es el encargado de coordinar el funcionamiento de la aplicación. Responde a los eventos del usuario para lo que hace peticiones al modelo (para obtener o cambiar la información) y a la vista (para que muestre al usuario dicha información).

Este patrón de arquitectura de software se basa en las ideas de reutilización de código y la separación de conceptos, características que buscan facilitar la tarea de desarrollo de aplicaciones y su posterior mantenimiento.

## Una aplicación sin MVC

Si una aplicación no utiliza este modelo, cuando una función modifica los datos debe además reflejar dicha modificación en la página para que la vea el usuario.

Por ejemplo: vamos a hacer una aplicación para gestionar un almacén. Entre otras muchas cosas tendrá una función (que podemos llamar **addProduct**) que se encargue de añadir un nuevo producto al almacén. Dicha función deberá realizar:

- añadir el nuevo producto al almacén (por ejemplo, añadiéndolo a un array de productos)
- pintar en la página ese nuevo producto (por ejemplo, añadiendo una nueva línea a una tabla donde se muestran los productos)

```
// La función que se ejecuta cuando el usuario envía el
// formulario para añadir un producto debería hacer:

// Coge los datos del formulario
const name = document.getElementById('product-form-name').value
const price = document.getElementById('product-form-name').price
...
// Valida cada dato
if (!name || name.length < 5 || ...)
...
// Añade el producto a la BBDD
const prod = await addProductToDatabase(payload)
let newProd = new Product(prod.id, prod.name, prod.price, prod.units)
this.products.push(newProd)
...
// Pinta en la página el nuevo producto
const DOMproduct = document.createElement('tr')
...
document.getElementById('products-table').appendChild(DOMproduct)
// Gestiona los posibles errores producidos en todo este proceso
...
```

Como vemos, se va a convertir en una función muy grande y que se encarga de muchas cosas distintas por lo que va a ser difícil mantener ese código. Además, toda la función es muy dependiente del HTML (en muchas partes se buscan elementos por su id).

## Nuestro patrón MVC

En una aplicación muy sencilla puede no seguirse este modelo, pero en cuanto la misma se complica un poco es imprescindible programar siguiendo buenas prácticas ya que en otro caso el código se volverá rápidamente muy difícil de mantener.

Hay muchas formas de implementar este modelo. Si se trata de un proyecto con OOP se debe seguir el patrón MVC usando clases. Si sólo se utiliza programación estructurada será igual, pero en vez de clases y métodos habrá funciones.

Para organizar el código se crean 3 subcarpetas dentro de la carpeta src:

- **model**: aquí incluiremos las clases que constituyen el modelo de nuestra aplicación
- **view**: aquí crearemos un fichero JS que será el encargado de la GUI de nuestra aplicación, el único dependiente del HTML. Este fichero será una clase que representa toda la vista, aunque en aplicaciones mayores lo normal es tener clases para cada página, etc.
- **controller**: aquí estará el fichero JS que contendrá el controlador de la aplicación

Al hacerlo así, para cambiar la forma en que se muestra algo, se hace directamente en la vista modificando la función que se ocupa de ello.

La vista será una clase cuyas propiedades serán elementos de la página HTML a los que se accederá frecuentemente, para no tener que buscarlos cada vez y para que estén disponibles para el controlador. Contendrá métodos para renderizar los distintos elementos de la vista.

El controlador será una clase cuyas propiedades serán el modelo y la vista, de forma que pueda acceder a ambos elementos. Tendrá métodos para las distintas acciones que pueda hacer el usuario (y que se ejecutarán como

respuesta a dichas acciones, tal como se explica en el tema de eventos). Cada uno de esos métodos llamará a métodos del modelo (para obtener o cambiar la información necesaria) y posteriormente de la vista (para reflejar esos cambios en lo que ve el usuario).

El fichero principal de la aplicación instanciará un controlador y lo inicializará.

Por ejemplo, siguiendo con la aplicación para gestionar un almacén. El modelo constará de:

- La clase **Store** que es nuestro almacén de productos (con métodos para añadir o eliminar productos, etc).
- La clase **Product** que gestiona cada producto del almacén (con métodos para crear un nuevo producto, etc).

El fichero principal sería algo como:

### main.js

```
const storeApp = new Controller()          // crea el controlador
storeApp.init()                          // lo inicializa

// En desarrollo podemos añadir algunas líneas que luego quitaremos para
// imitar acciones del usuario y así ver el funcionamiento de la aplicación:
storeApp.addProductToStore({ name: 'Portátil Acer Travelmate E2100', price: 523.12 })
storeApp.changeProduct({ id: 1, price: 515.95 })
storeApp.deleteProduct(1)
```

### model/store.class.js

```
export default class Store {
    constructor (id) {
        this.id=Number(id)
        this.products=[]
    }

    addProduct(payload) {
        // llama a la BBDD para que añada el producto
        const prod = await addProductToDatabase(payload)
        let newProd = new Product(prod.id, prod.name, prod.price, prod.units)
        this.products.push(newProd)
        return newProd
    }

    findProduct(id) {
        ...
    }
    ...
}
```

### model/product.class.js

```
export default class Product {
    constructor (id, name, price, units) {
```

```

        this.id = id
        this.name = name
        this.price = price
        this.units = units
    }
    ...
}

```

## view/index.js

```

export default class View {
    constructor {
        this.messageDiv = document.getElementById('messages')
        this.productForm = document.getElementById('product-form')
        this.productsList = document.getElementById('products-table')
    }

    init() {
        ...           // inicializa la vista, si es necesario
    }

    renderNewProduct(prod) {
        // código para añadir a la tabla el producto pasado añadiendo una nueva fila
        const DOMproduct = document.createElement('tr')
        ...
        this.productsList.appendChild(DOMproduct)
    }

    ...

    showMessage(type, message) {
        // código para mostrar mensajes al usuario y no tener que usar los alert
        const DOMmessage = document.createElement('div')
        ...
        this.messageDiv.appendChild(DOMmessage)
    }
}

```

## controller/index.js

```

export default class Controller {
    constructor() {
        this.store = new Store(1)          // crea el modelo, un Store con id 1
        this.view = new View()            // crea la vista
    }

    init() {
        this.view.init()                // inicializa la vista, si es necesario

        // Pone los escuchadores para poder interactuar con el usuario
        // Por ejemplo para enterarse de si el usuario envía el formulario:
    }
}

```

```
this.view.productForm.addEventListener('submit', (event) => {
    event.preventDefault()

    // Pide a la vista los datos del formulario
    const payload = this.view.getProductForm()
    // Y llama al método que se ocupa de añadir un producto
    this.addProductToStore(payload)
})

addProductToStore(prod) {
    // haría las comprobaciones necesarias sobre los datos y luego
    try {
        // dice al modelo que añada el producto
        const newProd = this.store.addProduct(prod)
        // si lo ha hecho le dice a la vista que lo pinte
        this.view.renderNewProduct(newProd)
    } catch(err) {
        this.view.showErrorMessage('error', 'Error al añadir el producto')
    }
}
...
}
```

Se puede obtener más información y ver un ejemplo más completo en <https://www.natapuntes.es/patron-mvc-en-vanilla-javascript/>



## DWEC – Javascript Web Cliente.

# DAW Cliente – MVC

El patrón Modelo-Vista-Controlador (MVC) es un enfoque de diseño de software que divide una aplicación en tres componentes principales: el Modelo, la Vista y el Controlador. Este patrón se puede aplicar tanto en el lado del cliente como en el lado del servidor. Aquí te proporcionaré una breve descripción de cómo puedes implementar el patrón MVC en el lado del cliente, principalmente utilizando JavaScript para la lógica del cliente y HTML/CSS para la presentación.

- Modelo (Model):** El modelo representa los datos y la lógica de la aplicación. Puede incluir funciones para acceder y modificar los datos.
- Vista (View):** La vista es la interfaz de usuario (UI). Representa la presentación de los datos y responde a las interacciones del usuario.
- Controlador (Controller):** El controlador maneja las interacciones del usuario y actualiza tanto el modelo como la vista en consecuencia.

Ejemplo sencillo de una aplicación web básica usando MVC:

### index.html

```
<!-- index.html (Vista) -->
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Mi Aplicación MVC</title>
    <style>
        /* Estilos CSS para la presentación */
        body {
            font-family: Arial, sans-serif;
            text-align: center;
        }
    </style>
</head>
<body>
    <h1>Mi Aplicación MVC</h1>
    <button id="b1">Cargar datos</button>
    <p id="output">Datos del Modelo: <span id="data"></span></p>
    <script type="module" src="controller.js"></script>
</body>
</html>
```

## model.js

```
// model.js (Modelo)
const model = {
    data: null,

    fetchData: async function() {
        return new Promise(resolve => {
            setTimeout(() => {
                this.data = 'Datos cargados desde el modelo';
                resolve();
            }, 1000);
        });
    }
};

// Exportar el modelo para que pueda ser importado por otros archivos
export default model;
```

## view.js

```
// view.js (Vista)
const view = {
    render: function(data) {
        document.getElementById('data').textContent = data;
    }
};

// Exportar la vista para que pueda ser importada por otros archivos
export default view;
```

## controller.js

```
// controller.js (Controlador)
import model from './model.js'; // Importar el modelo
import view from './view.js'; // Importar la vista

const controller = {
    loadData: async function() {
        await model.fetchData();
        view.render(model.data);
    }
};

document.getElementById('b1').addEventListener('click', controller.loadData)

// Exportar el controlador para que pueda ser importado por otros archivos
export default controller;

// Iniciar la aplicación cargando el controlador
```

```
//controller.loadData();
```

Este es un ejemplo básico que simula la carga de datos desde el modelo (usando un temporizador en lugar de una solicitud a una API real). Cada componente (Modelo, Vista y Controlador) tiene su propio archivo, pero en aplicaciones más grandes, puedes estructurarlos de manera diferente, como en carpetas separadas.

Al hacer clic en el botón "Cargar Datos", el controlador solicita al modelo cargar los datos, y luego actualiza la vista con los datos obtenidos del modelo. Este enfoque facilita la escalabilidad y el mantenimiento de tu código al separar las responsabilidades y permitir que cada componente realice su función específica.

En este ejemplo:

- En `model.js`, he utilizado `export default` para exportar el objeto `model`.
- En `view.js`, he utilizado `export default` para exportar el objeto `view`.
- En `controller.js`, he utilizado `import` para importar `model` y `view`.

Estas declaraciones permiten que los archivos se comuniquen entre sí y utilicen las funciones y objetos definidos en otros archivos. Esto sigue las convenciones de importación/exportación de módulos en JavaScript y es compatible tanto con navegadores modernos como con entornos Node.js.



## DWEC - Javascript Web Cliente.

# DAW Cliente - Corolario

Para realizar una aplicación web se debe tener:

Una api del lado servidor

Una api del lado cliente

Cada una de las apis debería ser creada a partir de clases.

### Lado servidor:

*rutas.js:*

```
const express = require('express');
const router = express.Router();
const UserController = require('./controllers/UserController');

const userController = new UserController();

router.get('/users', userController.getAllUsers.bind(userController));
router.get('/users/:id', userController.getUserById.bind(userController));
router.post('/users', userController.createUser.bind(userController));
router.put('/users/:id', userController.updateUser.bind(userController));
router.delete('/users/:id', userController.deleteUser.bind(userController));

module.exports = router;
```

*claseControladora.js*

```
class UserController {
  async getAllUsers(req, res) {
    // Lógica para obtener todos los usuarios
    // Puede acceder a la base de datos, realizar consultas, etc.
    const users = await User.find();
    res.json(users);
  }

  async getUserById(req, res) {
    const userId = req.params.id;
    // Lógica para obtener un usuario por ID
    const user = await User.findById(userId);
    if (!user) {
      return res.status(404).json({ error: 'Usuario no encontrado' });
    }
  }
}
```

```

    }
    res.json(user);
}

async createUser(req, res) {
  const { name, email, password } = req.body;
  // Lógica para crear un nuevo usuario
  const newUser = new User({ name, email, password });
  await newUser.save();
  res.status(201).json(newUser);
}

async updateUser(req, res) {
  const userId = req.params.id;
  const { name, email, password } = req.body;
  // Lógica para actualizar un usuario por ID
  const updatedUser = await User.findByIdAndUpdate(userId, { name, email, password }, { new: true });
  if (!updatedUser) {
    return res.status(404).json({ error: 'Usuario no encontrado' });
  }
  res.json(updatedUser);
}

async deleteUser(req, res) {
  const userId = req.params.id;
  // Lógica para eliminar un usuario por ID
  const deletedUser = await User.findByIdAndDelete(userId);
  if (!deletedUser) {
    return res.status(404).json({ error: 'Usuario no encontrado' });
  }
  res.json(deletedUser);
}

module.exports = UserController;

```

## Lado Cliente:

*apiCliente.js*

```

class ApiClient {
  constructor(baseURL) {
    this.baseURL = baseURL;
  }

  async fetchData(endpoint) {
    try {
      const response = await fetch(`${this.baseURL}/${endpoint}`);

```

```
        const data = await response.json();
        return data;
    } catch (error) {
        console.error('Error fetching data:', error);
        throw error;
    }
}

async postData(endpoint, body) {
    try {
        const response = await fetch(`.${this.baseURL}/${endpoint}`, {
            method: 'POST',
            headers: {
                'Content-Type': 'application/json',
            },
            body: JSON.stringify(body),
        });
        const data = await response.json();
        return data;
    } catch (error) {
        console.error('Error posting data:', error);
        throw error;
    }
}

async putData(endpoint, body) {
    try {
        const response = await fetch(`.${this.baseURL}/${endpoint}`, {
            method: 'PUT',
            headers: {
                'Content-Type': 'application/json',
            },
            body: JSON.stringify(body),
        });
        const data = await response.json();
        return data;
    } catch (error) {
        console.error('Error updating data:', error);
        throw error;
    }
}

async patchData(endpoint, body) {
    try {
        const response = await fetch(`.${this.baseURL}/${endpoint}`, {
            method: 'PATCH',
            headers: {
                'Content-Type': 'application/json',
            },
            body: JSON.stringify(body),
        });
        const data = await response.json();
    }
}
```

```
        return data;
    } catch (error) {
        console.error('Error patching data:', error);
        throw error;
    }
}

async deleteData(endpoint) {
    try {
        const response = await fetch(` ${this.baseURL}/${endpoint}` , {
            method: 'DELETE',
        });
        const data = await response.json();
        return data;
    } catch (error) {
        console.error('Error deleting data:', error);
        throw error;
    }
}

// Ejemplo de uso
const api = new ApiClient('https://tu-api-express.com');

// Utilizar los métodos según sea necesario
api.fetchData('ruta-api')
    .then(data => console.log('Datos recibidos:', data))
    .catch(error => console.error('Error al obtener datos:', error));

const dataToSend = { key: 'value' };

api.postData('ruta-api', dataToSend)
    .then(response => console.log('Respuesta del servidor (POST):', response))
    .catch(error => console.error('Error al enviar datos (POST):', error));

api.putData('ruta-api/1', dataToSend)
    .then(response => console.log('Respuesta del servidor (PUT):', response))
    .catch(error => console.error('Error al enviar datos (PUT):', error));

api.patchData('ruta-api/1', { updatedKey: 'updatedValue' })
    .then(response => console.log('Respuesta del servidor (PATCH):', response))
    .catch(error => console.error('Error al enviar datos (PATCH):', error));

api.deleteData('ruta-api/1')
    .then(response => console.log('Respuesta del servidor (DELETE):', response))
    .catch(error => console.error('Error al eliminar datos:', error));
```

## Mysql y async/await (necesita módulo mysql2 en express)

Se puede utilizar async/await para hacer consultas asíncronas a mysql (en node express requiere el módulo mysql2)

`mysql2` proporciona una versión basada en promesas de sus métodos, lo que facilita el uso de `async/await`. Es importante gestionar los errores correctamente y cerrar adecuadamente las conexiones o el pool de conexiones cuando sea necesario.

Por ejemplo, la función `getAllUsers` está marcada como `async`, y utilizamos `await pool.execute(query)` para ejecutar la consulta de forma asíncrona y esperar a que se complete antes de continuar con el código.

En `mysql2`, la función `execute` no retorna una Promesa directamente, pero puedes envolverla en una Promesa utilizando `util.promisify`. Aquí te dejo un ejemplo:

Primero, necesitarás importar el módulo `util` de Node.js para usar la función `promisify`:

```
const util = require('util');
const mysql = require('mysql2');

const pool = mysql.createPool({
  host: 'localhost',
  user: 'tu_usuario',
  password: 'tu_contraseña',
  database: 'tu_base_de_datos',
  waitForConnections: true,
  connectionLimit: 10,
  queueLimit: 0,
});

// Utilizar promisify para envolver la función execute
const executeAsync = util.promisify(pool.execute).bind(pool);

class UserController {
  async getAllUsers(req, res) {
    try {
      // Lógica para obtener todos los usuarios
      const query = 'SELECT * FROM users'; // Asegúrate de que tu tabla de usuarios se llama 'users'

      const [rows, fields] = await executeAsync(query);

      res.json(rows);
    } catch (error) {
      console.error('Error al obtener usuarios:', error);
      res.status(500).json({ error: 'Error interno del servidor' });
    }
  }
}

// ... otros métodos del controlador
}

module.exports = UserController;
```

En este ejemplo, hemos envuelto la función `execute` en una versión promisificada llamada `executeAsync`. Ahora, puedes utilizar `await executeAsync(query)` para ejecutar la consulta de forma asíncrona.

Esta es una manera de trabajar con `async/await` y `mysql2` cuando la función que necesitas usar no retorna directamente una Promesa.

*Lo mismo sin necesidad de pool en la base de datos:*

```
const util = require('util');
const mysql = require('mysql2');

// Configuración de la conexión a la base de datos
const connection = mysql.createConnection({
  host: 'localhost',
  user: 'tu_usuario',
  password: 'tu_contraseña',
  database: 'tu_base_de_datos',
});

// Utilizar promisify para envolver la función query
const queryAsync = util.promisify(connection.query).bind(connection);

class UserController {
  async getAllUsers(req, res) {
    try {
      // Lógica para obtener todos los usuarios
      const query = 'SELECT * FROM users'; // Asegúrate de que tu tabla de usuarios se llama 'users'

      const [rows, fields] = await queryAsync(query);

      res.json(rows);
    } catch (error) {
      console.error('Error al obtener usuarios:', error);
      res.status(500).json({ error: 'Error interno del servidor' });
    }
  }
}

// ... otros métodos del controlador
}

module.exports = UserController;
```

**Cerrar las conexiones a la base de datos:**

Para cerrar la conexión a la base de datos en `mysql2`, puedes utilizar el método `end`. Aquí tienes un ejemplo de cómo puedes hacerlo:

```
const mysql = require('mysql2');

// Configuración de la conexión a la base de datos
const connection = mysql.createConnection({
```

```

host: 'localhost',
user: 'tu_usuario',
password: 'tu_contraseña',
database: 'tu_base_de_datos',
});

// Lógica para obtener todos los usuarios
async function getAllUsers() {
  try {
    const query = 'SELECT * FROM users'; // Asegúrate de que tu tabla de usuarios se
    // llama 'users'

    const [rows, fields] = await connection.promise().query(query);

    console.log(rows);

    // Cerrar la conexión después de usarla
    connection.end();
  } catch (error) {
    console.error('Error al obtener usuarios:', error);
  }
}

// Llamada a la función
getAllUsers();

```

En este ejemplo, después de realizar la consulta y procesar los resultados, he agregado la línea `connection.end()` para cerrar la conexión.

Es importante destacar que debes cerrar la conexión después de haber realizado todas las operaciones necesarias en tu aplicación. Si tienes múltiples consultas o interacciones con la base de datos, es recomendable cerrar la conexión una vez que hayas terminado con todas las operaciones relacionadas con la base de datos.

Ten en cuenta que si estás utilizando un enfoque de conexión por solicitud (creando y cerrando conexiones para cada solicitud), esta práctica es más común. Sin embargo, si estás utilizando un pool de conexiones, generalmente no necesitas cerrar la conexión manualmente, ya que el pool maneja la administración de conexiones por ti.

## Uso de pool de conexión a MySQL (conectar y liberar)

*db.js*

```

const mysql = require('mysql2'); // se instala con: npm i mysql
const db = mysql.createPool({
  //propiedad host donde se ubica la base de datos Mysql
  host: '0.0.0.0',
  //port: 3307,      // se puede cambiar el puerto de mysql

  user: 'ejemplouser',
  //contraseña del usuario en mysql
  password: '123456',
  //nombre del esquema al que nos vamos a conectar
  database: 'ejemplodb',
}

```

```

waitForConnections: true,
connectionLimit: 10,
queueLimit: 0,
});
module.exports=db;

```

*ciudadesController.js*

```

const db = require('../databases/db');

const getCiudades= (req, res) => { //http://localhost:3000/ciudades
// Consulta a la base de datos
// Obtener una conexión del pool
db.getConnection((err, connection) => {
if (err) {
  console.error('Error al obtener la conexión del pool:', err);
} else {
  console.log('Conexión exitosa al pool de conexiones');

  // Realizar operaciones con la conexión
  // Ejemplo: Consulta a la base de datos
  connection.query('SELECT * FROM ciudades', (queryError, results) => {
    if (queryError) {
      console.error('Error en la consulta:', queryError);
    } else {
      console.log('Resultados de la consulta:', results);
      res.json(results)
    }
  }

  // Importante: Liberar la conexión de vuelta al pool cuando hayas terminado
  // con ella
  connection.release();
})
}
});

module.exports={
  getCiudades,
  crearCiudad,
  getCiudadById,
  putCiudad,
  patchCiudad,
  actualizarCiudad,
  deleteCiudad,
  getCiudadesByHabitantes,
};

```

## DWEC - Javascript Web Cliente.

# DWEC – Fichero de Configuración

En una aplicación web con Node.js, Express y MySQL, las constantes como la dirección IP del servidor Express, el puerto, la dirección IP de MySQL y el puerto de MySQL se suelen definir en un archivo de configuración o en un módulo separado para mantener un código más organizado y fácil de mantener.

Aquí hay un ejemplo de cómo podrías organizar tus constantes en un archivo de configuración (**config.js**):

```
// config.js

module.exports = {
  express: {
    host: '127.0.0.1', // Dirección IP del servidor Express
    port: 3000 // Puerto del servidor Express
  },
  mysql: {
    host: 'localhost', // Dirección IP de MySQL
    port: 3306, // Puerto de MySQL
    user: 'usuario_mysql',
    password: 'contraseña_mysql',
    database: 'nombre_de_la_base_de_datos'
  }
};
```

Luego, en tu archivo principal de la aplicación (por ejemplo, **app.js**), puedes importar estas constantes:

```
// app.js

const express = require('express');
const mysql = require('mysql');
const config = require('./config');

const app = express();

// Configuración de Express
app.listen(config.express.port, config.express.host, () => {
  console.log(`Servidor Express escuchando en
http://${config.express.host}:${config.express.port}`);
});

// Configuración de MySQL
const connection = mysql.createConnection({
  host: config.mysql.host,
```

```
port: config.mysql.port,  
user: config.mysql.user,  
password: config.mysql.password,  
database: config.mysql.database  
});  
  
// Resto de la configuración de tu aplicación...
```

Esta estructura te permite centralizar todas las configuraciones en un solo lugar, facilitando la modificación y el mantenimiento de tu aplicación. Además, puedes tener diferentes archivos de configuración para entornos de desarrollo, prueba y producción.



## DWEC - Javascript Web Cliente.

# DWEC – Páginas html

En una aplicación web basada en Node.js, Express y MySQL, generalmente no suele haber un único archivo HTML. Por lo general, las aplicaciones web tienen múltiples páginas o vistas, y cada una de ellas tiene su propio archivo HTML. Estos archivos HTML pueden estar organizados en un directorio específico, como por ejemplo un directorio llamado "views".

En una aplicación web típica, las rutas definidas en Express pueden renderizar diferentes vistas según las solicitudes del usuario. Puedes usar un motor de plantillas (template engine) como EJS, Pug, o Handlebars para generar HTML dinámicamente en el servidor y luego enviarlo al cliente.

Un ejemplo sencillo con Express y EJS podría verse así:

### 1. Organización de carpetas:

```
markdown Copy code
-
- proyecto
  - public
    - styles
      - style.css
  - views
    - index.ejs
    - otraVista.ejs
  - node_modules
  - app.js
  - package.json
```

## 1. Archivo `app.js`:

```
javascript Copy code  
  
const express = require('express');  
const app = express();  
const port = 3000;  
  
app.set('view engine', 'ejs');  
  
app.get('/', (req, res) => {  
    res.render('index');  
});  
  
app.get('/otraRuta', (req, res) => {  
    res.render('otraVista');  
});  
  
app.listen(port, () => {  
    console.log(`La aplicación está escuchando en http://local`)  
});
```

## 1. Archivo `views/index.ejs`:

```
html
Copy code

<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Mi Aplicación</title>
  <link rel="stylesheet" href="/styles/style.css">
</head>
<body>
  <h1>Bienvenido a mi aplicación</h1>
  <!-- Contenido de la página -->
</body>
</html>
```

Cada ruta en tu aplicación puede renderizar una vista diferente. La estructura exacta puede variar según las necesidades específicas de tu aplicación, pero generalmente se prefiere organizar los archivos HTML en directorios específicos para facilitar la administración del código fuente.

## Página de entrada a la aplicación

La página de entrada o página principal de una aplicación web suele ser determinada por la ruta definida en Express. En el ejemplo anterior, la página de entrada se establece en la ruta raíz ("/") mediante la siguiente declaración:

```
javascript
Copy code

app.get('/', (req, res) => {
  res.render('index');
});
```

En este caso, cuando un usuario accede a la URL principal de la aplicación (por ejemplo, <http://localhost:3000/>), se renderiza la vista llamada 'index.ejs'. Esta vista representa la página principal de la aplicación.

Sin embargo, ten en cuenta que la designación de la página de entrada puede variar según la estructura y la lógica de tu aplicación. Puedes configurar Express para que la página de entrada sea diferente o incluso dinámica según ciertas condiciones o la autenticación del usuario. La página de entrada es simplemente la primera página que se muestra al usuario cuando accede al sitio web.

## Enfoque SPA (Aplicación de página única)

Las "aplicaciones de página única" (SPA, por sus siglas en inglés), se refieren a un enfoque de desarrollo web en el que la interfaz de usuario se carga y actualiza dinámicamente en una sola página, en lugar de cargar páginas HTML completamente nuevas desde el servidor. Este enfoque proporciona una experiencia de usuario más fluida, ya que solo se cargan los datos necesarios y no toda la página.

Las SPAs suelen ser construidas utilizando frameworks o bibliotecas de JavaScript, como React, Angular o Vue.js. Estos frameworks permiten la creación de componentes reutilizables que se pueden actualizar de manera eficiente en respuesta a las interacciones del usuario sin recargar la página completa.

En el contexto de una SPA, la "página de entrada" se refiere a la primera vista que se carga cuando el usuario accede al sitio web. Esta vista puede incluir la estructura principal de la aplicación y cargar dinámicamente los datos necesarios. Sin embargo, en una SPA, el término "página de entrada" puede ser un poco menos relevante que en las aplicaciones tradicionales de varias páginas.

La elección entre el desarrollo de SPA y aplicaciones tradicionales de varias páginas depende de los requisitos específicos del proyecto y las preferencias del equipo de desarrollo. Ambos enfoques tienen sus ventajas y desventajas, y la elección dependerá de factores como la complejidad de la aplicación, la experiencia del usuario deseada y los recursos disponibles.



## DWEC - Javascript Web Cliente.

# DAW Cliente - Rutas RESTful

Las rutas **RESTful** son un enfoque de diseño de API (Interfaz de Programación de Aplicaciones) que sigue los principios de REST (Transferencia de Estado Representacional). REST es un conjunto de convenciones y restricciones para diseñar servicios web que utilizan el protocolo HTTP de manera coherente y eficiente.

En el contexto de las rutas RESTful, se refiere a la forma en que los recursos (entidades o datos) son expuestos a través de las URLs (Uniform Resource Locators). Las rutas siguen un patrón jerárquico y están diseñadas de manera lógica y fácil de entender. Algunas de las características clave de las rutas RESTful incluyen:

1. **Nombres de Recursos Significativos:** Las rutas deben representar recursos específicos en lugar de acciones. Por ejemplo, en lugar de tener una ruta como `/obtenerCliente`, debería ser `/clientes`.
2. **Uso de Métodos HTTP:** Las operaciones CRUD (Crear, Leer, Actualizar, Eliminar) son mapeadas a los métodos HTTP estándar. Por ejemplo:
  - GET para obtener información.
  - POST para crear nuevos recursos.
  - PUT o PATCH para actualizar recursos.
  - DELETE para eliminar recursos.
3. **Jerarquía y Anidamiento:** La jerarquía de las rutas debe reflejar la relación entre los recursos. Por ejemplo, `/clientes/123/productos` podría representar los productos asociados al cliente con el ID 123.
4. **Uso de Plurales:** Los nombres de recursos en las rutas generalmente se escriben en plural para representar colecciones de recursos. Por ejemplo, `/clientes` en lugar de `/cliente`.

## Ejemplo de rutas RESTful para un sistema de gestión de clientes:

- **GET /clientes:** Obtener la lista de todos los clientes.
- **GET /clientes/123:** Obtener detalles del cliente con ID 123.
- **POST /clientes:** Crear un nuevo cliente.
- **PUT /clientes/123:** Actualizar los detalles del cliente con ID 123.
- **DELETE /clientes/123:** Eliminar el cliente con ID 123.

Al seguir estas convenciones, se logra un diseño de API consistente, fácil de entender y que aprovecha las capacidades del protocolo HTTP de manera eficiente.



## DWEC – Javascript Web Cliente.

Pasos crear una API con node.js y express .....	1
Añadir rutas: .....	4
Guardar datos en BD de MySQL.....	5
Detener una aplicación Express en Node.js.....	7
Para no tener que reiniciar el servidor con cada cambio:.....	9
Ejemplo de las tablas: .....	10
El fichero app.js quedaría: .....	11
Vamos a poner las rutas en distintos ficheros de rutas:.....	13
App.js .....	13
Rutas-ciudades.js .....	14
Sacamos la lógica a ficheros controladores:.....	16
Sacar la conexión a mysql en archivo aparte .....	18
Servir archivos estáticos .....	21
CORS y Fetch desde otro equipo .....	23
No es necesario abrir el Puerto de express en firewall.....	26
Actuaciones con el puerto de MySql .....	28

## Pasos crear una API con node.js y express

Ejemplo de cómo estructurar una API utilizando Node.js y Express:

```
api/
    └── controllers/
        ├── userController.js
        └── ...
    └── models/
        ├── user.js
        └── ...
    └── routes/
        ├── userRoutes.js
        └── ...
└── app.js
```

En esta estructura, tenemos las siguientes carpetas:

**controllers:** Esta carpeta contendrá los controladores de la API. Cada controlador se encargará de manejar la lógica de negocio relacionada con un modelo específico.

**models:** Esta carpeta contendrá los modelos de la API. Cada modelo representa una entidad en la base de datos y define su estructura y comportamiento.

**routes:** Esta carpeta contendrá las definiciones de las rutas de la API. Cada archivo de ruta se encargará de manejar las solicitudes que lleguen a una ruta específica.

**app.js:** Este archivo será el punto de entrada de la aplicación. Aquí configurarás y montarás todas las partes de la API, como los middleware y las rutas.

Veamos los pasos desde VSC:

### 1. Inicializa tu proyecto:

Abre tu terminal y ejecuta los siguientes comandos para crear un nuevo proyecto y entrar en el directorio:

```
bash
```

 Copy code

```
mkdir mi-api  
cd mi-api  
npm init -y
```

### 2. Instala Express:

Ejecuta el siguiente comando para instalar Express:

```
bash
```

 Copy code

```
npm install express
```

3 Crea el archivo principal e la aplicación (app.js)

```
const express = require('express');  
const cors = require('cors');  
  
// Crear una instancia de Express  
const app = express();  
  
// Habilitar CORS para todas las rutas  
app.use(cors());  
  
// Resto de tu configuración de Express y rutas aquí...  
  
// Puerto en el que escuchará el servidor  
const puerto = 3000;  
  
// Iniciar el servidor y escuchar en el puerto especificado  
app.listen(puerto, () => {  
  console.log(`Servidor escuchando en el puerto ${puerto}`);  
});
```

**Importar módulos:** Importamos los módulos necesarios `express` y `cors` utilizando `require`.

javascript

 Copy code

```
const express = require('express');
const cors = require('cors');
```

**Crear instancia de Express:** Creamos una instancia de Express llamada `app`. Express es un framework web para Node.js que facilita la creación de aplicaciones web y APIs.

javascript

 Copy code

```
const app = express();
```

**Habilitar CORS:** Utilizamos el middleware `cors` para habilitar CORS en todas las rutas de nuestra aplicación. Esto permite solicitudes desde cualquier origen.

javascript

 Copy code

```
app.use(cors());
```

**Configuración de rutas:** Aquí es donde agregarías las rutas específicas de tu aplicación. Esto puede incluir rutas para manejar solicitudes GET, POST, PUT, DELETE, etc.

javascript

 Copy code

```
// Resto de tu configuración de Express y rutas aquí...
```

Deberías agregar tus rutas y lógica de manejo de solicitudes aquí, dependiendo de los endpoints y la funcionalidad que necesites en tu aplicación.

**Iniciar el servidor:** Finalmente, iniciamos el servidor y lo hacemos escuchar en un puerto específico. En este caso, el servidor escuchará en el puerto 3000.

```
javascript Copy code  
  
const puerto = 3000;  
app.listen(puerto, () => {  
  console.log(`Servidor escuchando en el puerto ${puerto}`)  
});
```

Cuando ejecutas tu aplicación, el servidor comenzará a escuchar en el puerto especificado, y verás el mensaje en la consola indicando que el servidor está en funcionamiento.

## Añadir rutas:

```
// Resto de tu configuración de Express y rutas aquí...  
// Middleware para parsear el cuerpo de las peticiones en formato JSON  
app.use(express.json());  
  
// Ruta de ejemplo  
app.get('/', (req, res) => {  
  res.send('¡Bienvenido a mi API!');  
});  
  
// Ruta para obtener datos  
app.get('/api/datos', (req, res) => {  
  // Simulación de datos  
  const datos = {  
    mensaje: 'Estos son tus datos',  
    usuario: 'ejemplo',  
    edad: 25  
  };  
  
  res.json(datos);  
});  
  
// Ruta para enviar datos  
app.post('/api/enviar', (req, res) => {  
  const { mensaje } = req.body;  
  
  // Puedes realizar operaciones con los datos recibidos  
  // En este caso, simplemente los devolvemos
```

```
res.json({ recibido: true, mensaje });
});
```

## Guardar datos en BD de MySQL

Para guardar datos en una base de datos MySQL desde tu aplicación Node.js con Express, primero necesitarás instalar el paquete `mysql` para Node.js. Puedes hacerlo ejecutando el siguiente comando:

```
bash
Copy code
npm install mysql
```

Luego, puedes modificar tu código para conectarte a la base de datos y guardar/recuperar datos. A continuación, te presento un ejemplo básico utilizando MySQL:

1. Instala el paquete MySQL:

```
bash
Copy code
npm install mysql
```

2. Modifica tu archivo `index.js`:

```
javascript
Copy code
const express = require('express');
const mysql = require('mysql');
const app = express();
const port = 3000;

// Middleware para parsear el cuerpo de las peticiones en formato JSON
app.use(express.json());
```

El index.js completo queda así:

```
const express = require('express');
const mysql = require('mysql');
const app = express();
const port = 3000;

// Middleware para parsear el cuerpo de las peticiones en formato JSON
app.use(express.json());

// Configuración de la conexión a la base de datos
```

```

const db = mysql.createConnection({
  host: 'localhost',
  user: 'tu_usuario_mysql',
  password: 'tu_contraseña_mysql',
  database: 'tu_base_de_datos'
});

// Conexión a la base de datos
db.connect((err) => {
  if (err) {
    console.error('Error de conexión a la base de datos:', err);
  } else {
    console.log('Conexión exitosa a la base de datos');
  }
});

// Ruta para obtener datos desde la base de datos
app.get('/api/datos', (req, res) => {
  // Consulta a la base de datos
  db.query('SELECT * FROM datos', (err, resultados) => {
    if (err) {
      console.error('Error al obtener datos desde la base de datos:', err);
      res.status(500).json({ error: 'Error interno del servidor' });
    } else {
      res.json(resultados);
    }
  });
});

// Ruta para enviar datos y guardar en la base de datos
app.post('/api/enviar', (req, res) => {
  const { mensaje } = req.body;

  // Insertar datos en la base de datos
  db.query('INSERT INTO datos (mensaje) VALUES (?)', [mensaje], (err, resultado) => {
    if (err) {
      console.error('Error al guardar datos en la base de datos:', err);
      res.status(500).json({ error: 'Error interno del servidor' });
    } else {
      res.json({ recibido: true, mensaje, id: resultado.insertId });
    }
  });
});

// Inicia el servidor
app.listen(port, () => {
  console.log(`La aplicación está escuchando en http://localhost:${port}`);
});

```

### 3. Crea una tabla en tu base de datos:

Antes de ejecutar tu aplicación, asegúrate de tener una base de datos creada y una tabla para almacenar los datos. Puedes ejecutar la siguiente consulta SQL en tu cliente MySQL o herramienta de administración para crear una tabla de ejemplo:

```
sql Copy code
CREATE TABLE datos (
    id INT AUTO_INCREMENT PRIMARY KEY,
    mensaje VARCHAR(255) NOT NULL
);
```

### 4. Ejecuta tu aplicación:

Inicia tu servidor con `node index.js`, y ahora debería guardar y recuperar datos desde la base de datos MySQL.

Este es un ejemplo básico, y en un entorno de producción, se recomienda manejar las conexiones y errores de manera más robusta, y considerar medidas de seguridad como el uso de consultas preparadas para prevenir inyecciones SQL. Además, asegúrate de proteger la información de conexión a la base de datos.

## Detener una aplicación Express en Node.js

Para detener una aplicación Express en Node.js, puedes utilizar el método `close()` del servidor que se crea al llamar a `listen()` en tu aplicación Express. Aquí hay un ejemplo de cómo hacerlo:

```
javascript
const express = require('express');
const app = express();
const port = 3000;

// Configura tus rutas y middleware aquí

const server = app.listen(port, () => {
  console.log(`La aplicación está escuchando en http://localhost:${port}`);
});

// Para detener la aplicación
// Puedes llamar a server.close() cuando sea necesario, por ejemplo, en el
// código de cierre de tu aplicación.

// Ejemplo de detener la aplicación después de 5 segundos
setTimeout(() => {
  console.log('Deteniendo la aplicación...');
  server.close();
}, 5000);
```

En este ejemplo, la aplicación Express se inicia llamando a `app.listen()`, y el servidor devuelto se almacena en la variable `server`. Luego, puedes llamar a `server.close()` para detener la aplicación.

Ten en cuenta que si estás ejecutando la aplicación dentro de un script o aplicación más grande, es posible que necesites gestionar la lógica de cierre de manera más robusta. Por ejemplo, podrías usar señales de sistema operativo para detectar solicitudes de cierre y cerrar la aplicación de manera controlada. Aquí hay un ejemplo utilizando `process.on` para manejar la señal `SIGINT` (Ctrl+C en la consola):

```
javascript
const express = require('express');
const app = express();
const port = 3000;

// Configura tus rutas y middleware aquí

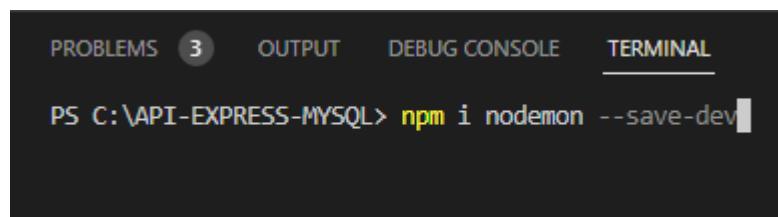
const server = app.listen(port, () => {
  console.log(`La aplicación está escuchando en http://localhost:${port}`);
});

// Manejar la señal SIGINT (Ctrl+C)
process.on('SIGINT', () => {
  console.log('Deteniendo la aplicación...');
  server.close(() => {
    console.log('La aplicación se ha detenido correctamente.');
    process.exit(0);
  });
});
```

Este enfoque te permite manejar de manera adecuada el cierre de la aplicación cuando se recibe la señal SIGINT.

## Para no tener que reiniciar el servidor con cada cambio:

Instalamos (abreviamos con npm i) el módulo **nodemon** con la opción --save-dev para que lo instale como “una dependencia de desarrollo” y así solo se tenga en cuenta durante el desarrollo de la app, y no cuando esté en producción.



The screenshot shows the VS Code interface with the 'TERMINAL' tab selected. The terminal window displays the command 'PS C:\API-EXPRESS-MYSQL> npm i nodemon --save-dev'. The command has been partially typed, with the rest of the line visible as a placeholder.

Estos cambios los veremos en el archivo package.json

En `package.json` realizamos otros cambios:

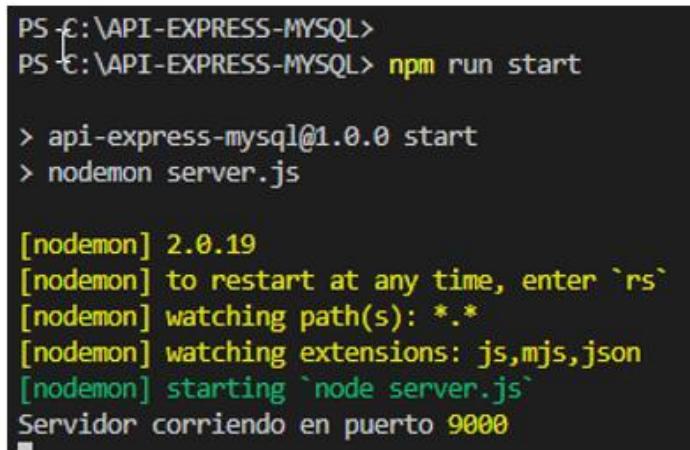
```
{
  "name": "api-express-mysql",
  "version": "1.0.0",
  "description": "",
  // cambiamos main por el nombre del archivo de entrada (server.js)
  "main": "server.js",
  "scripts": {
    // "test": "echo \"Error: no test specified\" && exit 1"

    // Dentro de scripts ponemos el nombre start(podría ser otro) y el
    // nombre del archivo de entrada (server.js)
    "start": "nodemon server.js"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "express": "^4.18.1"
  }
}
```

Añadimos en “scripts” la línea:

`“start”: “modemon server.js”`

Podemos poner otro nombre distinto a start, pero es el que nos va a servir para ejecutar el servidor con el comando:  
`npm run start`



```

PS C:\API-EXPRESS-MYSQL>
PS C:\API-EXPRESS-MYSQL> npm run start

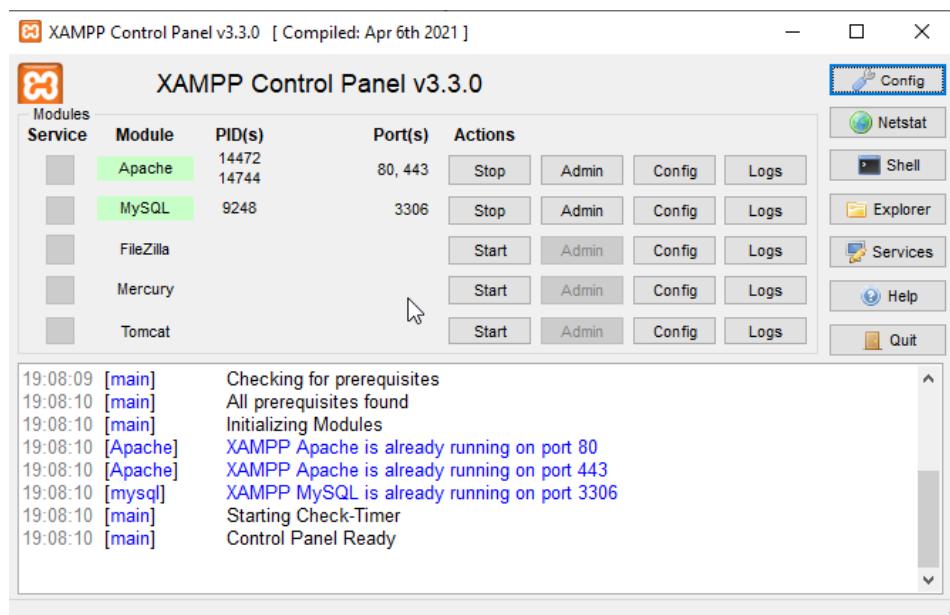
> api-express-mysql@1.0.0 start
> nodemon server.js

[nodemon] 2.0.19
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): ***!
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node server.js`
Servidor corriendo en puerto 9000
  
```

A partir de ahora cada vez que guardamos cambios (Ctrl+S) en el fichero .js, el servidor se reinicia solo.

## Ejemplo de las tablas:

Tenemos XAMPP e con Apache y MySQL arrancados



Podemos utilizar phpMyAdmin para administrar las tablas SQL, y los usuarios que tendrán acceso.

The screenshot shows a browser window with two tabs: 'localhost / 127.0.0.1 / ejemplod' and 'localhost:3001/api/reyes'. The main content is the phpMyAdmin interface for the 'ejemplodb' database. The 'Servidor' is set to '127.0.0.1' and the 'Base de datos' is 'ejemplodb'. The 'Estructura de tabla' tab is active. The left sidebar shows 'Reciente' and 'Favoritas' with a 'Nueva' button. The top navigation bar includes links for 'Examinar', 'Estructura', 'SQL', and 'Buscar'.

Es importante señalar que deberemos tener un usuario (distinto de root o del administrador MySQL) para acceder a la API.

Lo creamos en phpMyAdmin y asignamos los permisos a la base de datos de nuestra API.

Para nuestro ejemplo usaremos las tablas de la BD “ejemplodb” (ciudades, cuencas, embalses, paises, reyes)

The screenshot shows the 'Privilegios' (Privileges) section for the user 'ejemplouser'@'%' on the 'ejemplodb' database. The 'Base de datos' is 'ejemplodb', 'Privilegios' is 'ALL PRIVILEGES', 'Conceder' is 'Sí', and 'Privilegios específicos para la tabla' is 'No'. Below this, there are buttons for 'Editar privilegios' and 'Revocar'.

En este caso tenemos el usuario “ejemplouser” con todos los privilegios para la base de datos “ejemplodb”. Además es accesible desde cualquier IP -> ‘ejemplouser@%’

## El fichero app.js quedaría:

```

const express = require('express'); // se instala con: npm i express
const mysql = require('mysql'); // se instala con: npm i mysql
const app = express(); // esta es la aplicación que iniciamos
const port = 3000; // puerto para express

```

```
//EL PUERTO DEL SERVIDOR
// vamos a "setear" a través del método set de nuestra aplicación: ponemos la propiedad
//del puerto (port), y el valor: 3000
//app.set('port', 3000);
//mejoramos la línea anterior para que pueda usar el puerto que asigne el HOSTING cuando
//esté en producción, o el 3000.
app.set('port', process.env.PORT || port); // 'port' es una propiedad de app.set

// Middleware para parsear el cuerpo de las peticiones en formato JSON
app.use(express.json()); //middleware para JSON

// Configuración de la conexión a la base de datos
const db = mysql.createConnection({
  //propiedad host donde se ubica la base de datos Mysql
  host: '127.0.0.1',
  // port: 3306,      // se puede cambiar el puerto de mysql

  //usuario base de datos (por defecto sería root, pero hay que evitarlo)
  user: 'ejemplouser',
  //contraseña del usuario en mysql
  password: '123456',
  //nombre del esquema al que nos vamos a conectar
  database: 'ejemplodb'
});

// Conexión a la base de datos
db.connect((err) => {
  if (err) {
    console.error('Error de conexión a la base de datos:', err);
  } else {
    console.log('Conexión exitosa a la base de datos');
  }
});

// Ruta para obtener datos desde la base de datos
app.get('/api/datos', (req, res) => {
  // Consulta a la base de datos
  db.query('SELECT * FROM paises', (err, resultados) => {
    if (err) {
      console.error('Error al obtener datos desde la base de datos:', err);
      res.status(500).json({ error: 'Error interno del servidor' });
    } else {
      res.json(resultados);
    }
  });
});

// Ruta para enviar datos y guardar en la base de datos
app.post('/api/enviar', (req, res) => {
  const { mensaje } = req.body;

  // Insertar datos en la base de datos

```

```

db.query('INSERT INTO datos (mensaje) VALUES (?)', [mensaje], (err, resultado) => {
  if (err) {
    console.error('Error al guardar datos en la base de datos:', err);
    res.status(500).json({ error: 'Error interno del servidor' });
  } else {
    res.json({ recibido: true, mensaje, id: resultado.insertId });
  }
});

// Inicia el servidor
// Hacemos que el servidor empiece a escuchar en el puerto 9000 (por defecto en express)
// se hace con el método listen pasándole el número de puerto y una función
// que nos diga si está funcionando
/*
// versión antigua
app.listen(app.get('port'), ()=>{
  console.log("Servidor corriendo en puerto", app.get('port'));
})
*/
app.listen(port, () => {
  console.log(`La aplicación está escuchando en http://localhost:${port}`);
});

```

## Vamos a poner las rutas en distintos ficheros de rutas:

Quedando de momento 2 archivos:

### App.js

```

//requiero rutas (pongo un nombre cualquiera) que he exportado en routes.js
const rutasCiudades= require('./routes/rutas_ciudades');//no necesita extensión .js
//Ahora el servidor tiene un link a las rutas de rutas-ciudades.js
//necesita el require de rutes exportado

const express = require('express'); // se instala con: npm i express

const app = express(); // esta es la aplicación que iniciamos
const port = 3000; // puerto para express

//EL PUERTO DEL SERVIDOR
// vamos a "setear" a través del método set de nuestra aplicación: ponemos la propiedad
// del puerto (port), y el valor: 3000
//app.set('port', 3000);
//mejoramos la línea anterior para que pueda usar el puerto que asigne el HOSTING cuando
//esté en producción, o el 3000.
app.set('port', process.env.PORT || port); // 'port' es una propiedad de app.set

// Middleware para parsear el cuerpo de las peticiones en formato JSON
app.use(express.json()); //middleware para JSON

```

```
// Middleware para parsear el cuerpo de las peticiones en formato JSON
//app.use(bodyParser.json());

app.use('/ciudades', rutasCiudades); //así /ciudades es la raíz en rutas_ciudades.js

// Inicia el servidor
// Hacemos que el servidor empiece a escuchar en el puerto 9000 (por defecto en express)
// se hace con el método listen pasándole el número de puerto y una función
// que nos diga si está funcionando
/*
// versión antigua
app.listen(app.get('port'), ()=>{
  console.log("Servidor corriendo en puerto", app.get('port'));
})
*/
app.listen(port, () => {
  console.log(`La aplicación está escuchando en http://localhost:${port}`);
});
```

## Rutas-ciudades.js

```
// fichero de rutas: rutas_ciudades.js
//creo la constante para utilizar el módulo expresss que es requerido
const express= require('express');

// constante para utilizar el método Router() que se necesita
// y que debemos exportar
const rutasCiudades = express.Router();

// Configuración de la conexión a la base de datos
const mysql = require('mysql'); // se instala con: npm i mysql
const db = mysql.createConnection({
  //propiedad host donde se ubica la base de datos Mysql
  host: '127.0.0.1',
  // port: 3306,      // se puede camiar el puerto de mysql

  //usuario base de datos (por defecto sería root, pero hay que evitarlo)
  user: 'ejemplouser',
  //contraseña del usuario en mysql
  password: '123456',
  //nombre del esquema al que nos vamos a conectar
  database: 'ejemplodb'
});

// Conexión a la base de datos
db.connect((err) => {
```

```

if (err) {
  console.error('Error de conexión a la base de datos:', err);
} else {
  console.log('Conexión exitosa a la base de datos');
}

});

// aquí van todas las rutas que queramos

// Ruta para obtener datos desde la base de datos
  //cambiamos app por la nueva constante rutasCategorias
rutasCiudades.get('/', (req, res) => { //http://localhost:3000/ciudades
  // Consulta a la base de datos
  db.query('SELECT * FROM ciudades', (err, resultados) => {
    if (err) {
      console.error('Error al obtener datos desde la base de datos:', err);
      res.status(500).json({ error: 'Error interno del servidor' });
    } else {
      res.json(resultados);
    }
  });
});

// Ruta para enviar datos y guardar en la base de datos
rutasCiudades.post('/enviar', (req, res) => { //http://localhost:3000/ciudades/enviar
  const { nombre, cantidad } = req.body;

  // Insertar datos en la base de datos
  db.query('INSERT INTO ciudades (nombre, cantidad) VALUES (?, ?)', [nombre, cantidad], (err, resultado) => {
    if (err) {
      console.error('Error al guardar datos en la base de datos:', err);
      res.status(500).json({ error: 'Error interno del servidor' });
    } else {
      res.json({ recibido: true, nombre, cantidad, id: resultado.insertId });
    }
  });
});

// Ruta para obtener un registro por ID
rutasCiudades.get('/:id', (req, res) => { //http://localhost:3000/ciudades/registro/3
  const idRegistro = req.params.id;

  // Consulta a la base de datos para obtener el registro por ID
  db.query('SELECT * FROM ciudades WHERE id = ?', [idRegistro], (err, resultados) => {
    if (err) {
      console.error('Error al obtener el registro desde la base de datos:', err);
      res.status(500).json({ error: 'Error interno del servidor' });
    } else {
      // Verifica si se encontró un registro
      if (resultados.length > 0) {

```

```

        res.json(resultados[0]); // Devuelve el primer resultado encontrado (debería
    ser único)
    } else {
        res.status(404).json({ error: 'Registro no encontrado' });
    }
}
});

//exportamos la constante routes
module.exports = rutasCiudades;

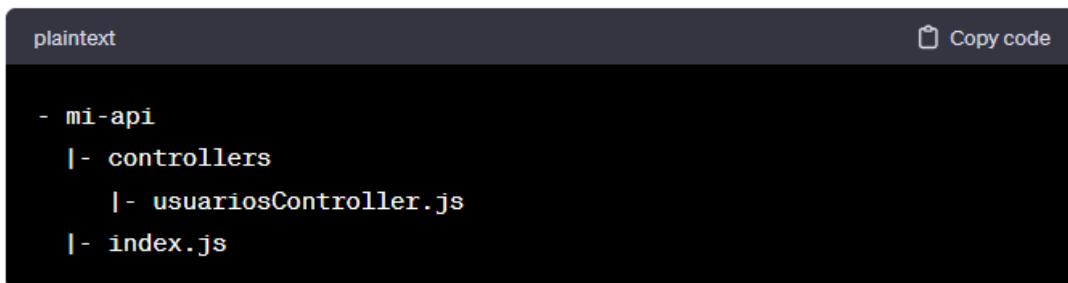
```

## Sacamos la lógica a ficheros controladores:

En Express, es una buena práctica organizar tu código en controladores para manejar las distintas funciones o rutas de tu API. Aquí tienes un ejemplo básico de cómo podrían verse los controladores en una aplicación Express y Node.js.

Supongamos que tienes una entidad llamada "Usuario" y quieras realizar operaciones CRUD (Crear, Leer, Actualizar, Eliminar) en tu API. Aquí hay un ejemplo con controladores:

1. Estructura del proyecto:



```

plaintext
Copy code

- mi-api
  |- controllers
    | - usuariosController.js
  |- index.js

```

## 2. Crea el controlador para Usuarios (`usuariosController.js`):

```
javascript                                     ⌂ Copy code

// controllers/usuariosController.js
const getAllUsuarios = (req, res) => {
    // Lógica para obtener todos los usuarios de la base de datos
    res.json({ mensaje: 'Obtener todos los usuarios' });
};

const getUsuarioById = (req, res) => {
    const userId = req.params.id;
    // Lógica para obtener un usuario por ID de la base de datos
    res.json({ mensaje: `Obtener usuario por ID: ${userId}` });
};

const createUsuario = (req, res) => {
    const nuevoUsuario = req.body;
    // Lógica para crear un nuevo usuario en la base de datos
    res.json({ mensaje: 'Crear un nuevo usuario', usuario: nuevoUsuario });
};

const updateUsuario = (req, res) => {

const updateUsuario = (req, res) => {
    const userId = req.params.id;
    const datosActualizados = req.body;
    // Lógica para actualizar un usuario por ID en la base de datos
    res.json({ mensaje: `Actualizar usuario por ID: ${userId}`, datos: datosActualizados });
};

const deleteUsuario = (req, res) => {
    const userId = req.params.id;
    // Lógica para eliminar un usuario por ID de la base de datos
    res.json({ mensaje: `Eliminar usuario por ID: ${userId}` });
};

module.exports = {
    getAllUsuarios,
    getUsuarioById,
    createUsuario,
    updateUsuario,
    deleteUsuario
};
```

### 3. Usa los controladores en tu aplicación (`index.js`):

```
javascript
// index.js
const express = require('express');
const bodyParser = require('body-parser');
const usuariosController = require('./controllers/usuariosController')

const app = express();
const port = 3000;

// Middleware para parsear el cuerpo de las peticiones en formato JSON
app.use(bodyParser.json());

// Rutas que utilizan los controladores
app.get('/api/usuarios', usuariosController.getAllUsuarios);
app.get('/api/usuarios/:id', usuariosController.getUsuarioById);
app.post('/api/usuarios', usuariosController.createUser);
app.put('/api/usuarios/:id', usuariosController.updateUser);
app.delete('/api/usuarios/:id', usuariosController.deleteUser);

// Inicia el servidor
app.listen(port, () => {
  console.log(`La aplicación está escuchando en http://localhost:${port}`);
});
```

En este ejemplo, cada función del controlador maneja una operación específica para la entidad "Usuario". Luego, en el archivo principal (`index.js`), se utilizan las funciones del controlador para manejar las rutas correspondientes de la API.

Esta estructura facilita la organización y mantenimiento del código, especialmente a medida que la aplicación crece en complejidad. Puedes expandir y ajustar según las necesidades específicas de tu aplicación.

## Sacar la conexión a mysql en archivo aparte

En una estructura de API con archivos de controladores y archivos de rutas en Express, es preferible realizar la conexión a MySQL en un archivo separado, como por ejemplo en un módulo de base de datos. Esto ayuda a mantener una estructura más organizada y modular, lo que facilita el mantenimiento y la escalabilidad de tu aplicación.

Aquí hay un ejemplo de cómo podrías estructurar tu aplicación:

## 1. Estructura del proyecto:

```
plaintext Copy code  
  
- mi-api  
| - controllers  
| | - usuariosController.js  
| - routes  
| | - usuariosRoutes.js  
| - database  
| | - db.js  
| - index.js
```

## 2. Crea el archivo de conexión a la base de datos (`database/db.js`):

```
javascript Copy code  
  
// database/db.js  
const mysql = require('mysql');  
  
const db = mysql.createConnection({  
    host: 'localhost',  
    user: 'tu_usuario_mysql',  
    password: 'tu_contraseña_mysql',  
    database: 'tu_base_de_datos'  
});  
  
db.connect((err) => {  
    if (err) {  
        console.error('Error de conexión a la base de datos:', err);  
    } else {  
        console.log('Conexión exitosa a la base de datos');  
    }  
});  
  
module.exports = db;
```

**3. Modifica el controlador y las rutas para utilizar la conexión a la base de datos**

(`controllers/usuariosController.js` y `routes/usuariosRoutes.js`):

```
javascript
// controllers/usuariosController.js
const db = require('../database/db');

const getAllUsuarios = (req, res) => {
    // Lógica para obtener todos los usuarios de la base de datos utilizando la conexión db
    res.json({ mensaje: 'Obtener todos los usuarios' });
};

// Resto de las funciones del controlador

module.exports = {
    getAllUsuarios,
    // Resto de las funciones del controlador
};
```

javascript

 Copy code

```
// routes/usuariosRoutes.js
const express = require('express');
const usuariosController = require('../controllers/usuariosController')

const router = express.Router();

// Rutas que utilizan los controladores
router.get('/', usuariosController.getAllUsuarios);
// Resto de las rutas

module.exports = router;
```

#### 4. Usa los archivos de rutas en tu aplicación principal (`index.js`):

```
javascript                                         Copy code

// index.js
const express = require('express');
const bodyParser = require('body-parser');
const usuariosRoutes = require('./routes/usuariosRoutes');

const app = express();
const port = 3000;

// Middleware para parsear el cuerpo de las peticiones en formato JSON
app.use(bodyParser.json());

// Rutas que utilizan los archivos de rutas
app.use('/api/usuarios', usuariosRoutes);
// Resto de las rutas

// Inicia el servidor
app.listen(port, () => {
  console.log(`La aplicación está escuchando en http://localhost:${port}`);
});
```

Con esta estructura, la conexión a la base de datos está encapsulada en un módulo separado (`database/db.js`), lo que facilita la gestión de la conexión y su reutilización en diferentes partes de la aplicación. Además, los controladores y las rutas pueden simplemente requerir este módulo para acceder a la conexión a la base de datos cuando sea necesario.

## Servir archivos estáticos

Para que la ruta raíz (` `/ `) de tu aplicación Express sirva una página HTML, puedes utilizar el middleware `express.static` para servir archivos estáticos y enviar el archivo HTML como respuesta. A continuación, te mostraré un ejemplo sencillo:

## 1. Estructura del proyecto:

```
plaintext Copy code  
  
- mi-api  
  |- public  
    |- index.html  
  |- index.js
```

En este ejemplo, el archivo `index.html` se encuentra en la carpeta `public` y contiene el contenido de tu página HTML.

## 2. Contenido de `index.html`:

```
html Copy code  
  
<!-- public/index.html -->  
<!DOCTYPE html>  
<html lang="es">  
  <head>  
    <meta charset="UTF-8">  
    <meta name="viewport" content="width=device-width, initial-scale=1">  
    <title>Página de Inicio</title>  
  </head>  
  <body>  
    <h1>Bienvenido a mi página de inicio</h1>  
    <p>Esta es una página de ejemplo.</p>  
  </body>  
</html>
```

### 3. Contenido de `index.js`:

```
javascript
Copy code

// index.js
const express = require('express');
const path = require('path');

const app = express();
const port = 3000;

// Middleware para servir archivos estáticos desde la carpeta 'public'
app.use(express.static(path.join(__dirname, 'public')));

// Ruta raíz que sirve la página HTML
app.get('/', (req, res) => {
  res.sendFile(path.join(__dirname, 'public', 'index.html'));
});

// Inicia el servidor
app.listen(port, () => {
  console.log(`La aplicación está escuchando en http://localhost:${port}`);
})
```

- `express.static`: Este middleware sirve archivos estáticos, como archivos HTML, CSS, JavaScript, etc., desde el directorio especificado (`public` en este caso). Se utiliza para servir los archivos estáticos de la página HTML.
- `app.get('/')`: Esta ruta maneja las solicitudes GET a la ruta raíz (`/`). Cuando un usuario accede a la ruta raíz, se envía el archivo HTML como respuesta utilizando `res.sendFile`.
- `path.join(\_\_dirname, 'public', 'index.html')`: Se utiliza para construir la ruta completa al archivo HTML. `\_\_dirname` representa el directorio actual del archivo en el que se está ejecutando el código.

Ahora, cuando ejecutas tu aplicación y accedes a `http://localhost:3000/`, deberías ver la página HTML en la ruta raíz. Ajusta los nombres de archivos y carpetas según la estructura específica de tu proyecto.

## CORS y Fetch desde otro equipo

Cuando estás haciendo una solicitud `fetch` a una API en otro equipo (dominio), es posible que te encuentres con problemas de política de mismo origen (Same-Origin Policy). La política de mismo origen es una política de seguridad en los navegadores web que restringe cómo los scripts en una página web pueden hacer solicitudes a otra página.

Para permitir solicitudes entre dominios diferentes, debes configurar el servidor para incluir encabezados CORS (Cross-Origin Resource Sharing). Estos encabezados le dicen al navegador que es seguro permitir solicitudes desde un dominio específico.

A continuación, te mostraré cómo configurar CORS en un servidor Express y cómo hacer una solicitud `fetch` desde el lado del cliente:

## Configuración de CORS en Express

Instala el paquete `cors` en tu proyecto:

bash

 Copy code

```
npm install cors
```

Luego, en tu servidor Express (`index.js`), agrega la configuración de CORS:

javascript

 Copy code

```
// index.js
const express = require('express');
const cors = require('cors');
const path = require('path');

const app = express();
const port = 3000;

// Configurar CORS
app.use(cors());

// Resto del código...
```

## Lado del Cliente (HTML y JavaScript)

En tu archivo HTML (`public/index.html`), puedes tener un botón o algún evento que desencadene la solicitud `fetch`:



The screenshot shows a code editor window with a dark theme. The file is named 'index.html'. The code is as follows:

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Solicitud Fetch a API Externa</title>
</head>
<body>
  <button id="btnFetch">Hacer solicitud Fetch</button>

  <script src="js/hacerFetch.js"></script>
</body>
</html>
```

In the code, there is a button with the ID 'btnFetch' and a script tag that points to 'js/hacerFetch.js'.

En tu archivo JavaScript (`public/js/hacerFetch.js`), puedes hacer la solicitud `fetch`:

```
javascript
// public/js/hacerFetch.js

document.getElementById('btnFetch').addEventListener('click', () => {
    // Reemplaza la URL con la URL de la API externa que deseas acceder
    const apiUrl = 'https://api.externa.com/data';

    fetch(apiUrl)
        .then(response => {
            if (!response.ok) {
                throw new Error(`Error de red - ${response.status}`);
            }
            return response.json();
        })
        .then(data => {
            console.log('Datos recibidos:', data);
            // Haz algo con los datos recibidos
        })
        .catch(error => console.error('Error en la solicitud Fetch:', error));
});
```

Asegúrate de reemplazar `https://api.externa.com/data` con la URL real de la API externa a la que estás intentando acceder.

Ten en cuenta que algunos servidores también pueden requerir autenticación, como claves API, tokens de acceso, etc. Asegúrate de seguir la documentación de la API externa para obtener detalles sobre cómo autenticarte correctamente si es necesario.

## No es necesario abrir el Puerto de express en firewall



You

por qué no es necesario abrir el puerto express para que se consuma la api desde otro equipo?



## ChatGPT

Cuando creas una API con Express y la expones en un servidor, por defecto, Express escucha en un puerto específico en la máquina donde se está ejecutando. Este puerto es donde la aplicación Express acepta solicitudes HTTP entrantes. Sin embargo, para que la API sea accesible desde otros equipos, no es necesario "abrir el puerto" en el sentido de abrir puertos en un firewall. La accesibilidad desde otros equipos depende de varios factores:

### 1. Dirección IP y Puerto:

- Si tu servidor Express está escuchando en una dirección IP específica (por ejemplo, `127.0.0.1` para conexiones solo locales) o en todas las interfaces (`0.0.0.0`), afectará si la API es accesible desde otros equipos.
- Si estás ejecutando Express en `localhost` o `127.0.0.1`, la API solo será accesible localmente. Para que sea accesible desde otros equipos, debes configurar Express para escuchar en la dirección IP de tu máquina o en `0.0.0.0` (todas las interfaces).

### 2. Firewall del Servidor:

- Asegúrate de que el firewall en el servidor permita conexiones al puerto en el que Express está escuchando. Si el puerto está bloqueado, las solicitudes no podrán llegar a tu aplicación.

### 3. Firewall del Router:

- Si estás detrás de un router, asegúrate de que el puerto en el que Express está escuchando esté redirigido correctamente desde el router al servidor. Esto se llama reenvío de puertos y es esencial para permitir que las solicitudes externas lleguen al servidor.

### 4. Configuración CORS:

- En el lado del servidor (Express), asegúrate de que has configurado CORS adecuadamente para permitir solicitudes desde dominios específicos o cualquier origen (`\*`). Esto es necesario para evitar problemas de política de mismo origen en los navegadores web.



En resumen, para que una API Express sea accesible desde otros equipos, debes asegurarte de que la aplicación esté escuchando en la dirección IP correcta, que el puerto esté abierto en el firewall del servidor y, si es necesario, configurar el reenvío de puertos en el router. También, asegúrate de que CORS esté configurado adecuadamente para permitir solicitudes desde los dominios deseados.

# Actuaciones con el puerto de MySql

**You**

¿Por qué no hay que abrir el puerto de mysql si está en el mismo servidor que express?

**ChatGPT**

Cuando tu aplicación Express y tu base de datos MySQL están en el mismo servidor, no necesitas "abrir el puerto" de MySQL para acceder desde la aplicación Express en el mismo servidor. Esto se debe a que las conexiones locales a la base de datos generalmente no pasan por el mismo proceso de enrutamiento que las conexiones externas.

^

Cuando tu aplicación Express y tu base de datos MySQL están en el mismo servidor, pueden comunicarse a través de conexiones locales (localhost o 127.0.0.1). En este caso, la comunicación se produce a través de un socket Unix o una conexión de bucle de retroalimentación, y no pasa por la red física. Por lo tanto, no es necesario abrir un puerto específico para MySQL en el firewall del servidor.

Si tu aplicación Express y tu base de datos MySQL están en servidores separados, entonces necesitarás configurar MySQL para permitir conexiones remotas y abrir el puerto correspondiente en el firewall del servidor MySQL para que las conexiones externas sean posibles.

## Error tipo MIME: al servir Node.js y express archivos.js

El error "Failed to load module script: Expected a JavaScript module script but the server responded with a MIME type of 'text/html'" generalmente indica que el servidor está devolviendo el tipo MIME 'text/html' en lugar de 'application/javascript' para el archivo JavaScript que estás tratando de cargar como un módulo.

Aquí hay algunas sugerencias para solucionar este problema:

### 1. Encabezado Content-Type en el servidor:

Asegúrate de que el servidor esté configurado para enviar el encabezado `Content-Type: application/javascript` para los archivos JavaScript. Esto puede variar según el servidor web que estés utilizando.

Si estás utilizando Node.js con Express, por ejemplo, puedes configurarlo así:

```
javascript
Copy code

app.use('/ruta-de-tus-archivos-js', (req, res, next) => {
  res.type('application/javascript');
  next();
});
```

Ajusta `'/ruta-de-tus-archivos-js'` según tu configuración.

### 2. Extensión del archivo:

Asegúrate de que el archivo JavaScript tenga la extensión `'.js'`. Los archivos que actúan como módulos deben tener una extensión de archivo válida para que el navegador los interprete correctamente.

### 3. Rutas relativas o absolutas:

Asegúrate de que la ruta al archivo JavaScript sea correcta y esté especificada de manera adecuada, ya sea relativa o absoluta.

Ejemplo de ruta relativa:

```
html
Copy code

<script type="module" src="./ruta/al/archivo.js"></script>
```

Ejemplo de ruta absoluta:

html

 Copy code

```
<script type="module" src="/ruta/al/archivo.js"></script>
```

Ajusta la ruta según la estructura de tu proyecto.

**4. Caché del navegador:**

Intenta borrar la caché del navegador o prueba en un modo de navegación privada/incógnito. A veces, el navegador puede almacenar en caché el tipo MIME incorrecto para un archivo.

Si después de revisar estas sugerencias el problema persiste, sería útil ver cómo estás sirviendo tus archivos JavaScript desde el servidor y cómo estás configurando las rutas y el tipo MIME.



## DWEC - Javascript Web Cliente.

# HTML: disabled o readonly

La diferencia principal entre las propiedades `disabled` y `readonly` en HTML es su efecto en la interacción del usuario con un elemento de entrada (como un `input`). Aquí hay una explicación de ambas:

### 1. `disabled`:

- Cuando se establece la propiedad `disabled` en un elemento de entrada, como un `<input>` o un `<button>`, el elemento se desactiva.
- Un elemento desactivado no se puede modificar ni seleccionar, y no responde a eventos de clic u otros eventos de usuario.
- En el caso de un `<input>`, el valor del elemento desactivado no se enviará junto con el formulario cuando se envíe.

Ejemplo:

```
<input type="text" readonly />
```

### 2. `readonly`:

- Cuando se establece la propiedad `readonly` en un `<input>`, el elemento se vuelve de solo lectura.
- Aunque el usuario puede ver el contenido del elemento, no puede modificarlo directamente.
- Sin embargo, el elemento aún responde a eventos de clic y otros eventos de usuario.
- En el caso de un `<input>`, el valor del elemento `readonly` se enviará junto con el formulario cuando se envíe.

Ejemplo:

```
<input type="text" readonly />
```

En resumen, `disabled` desactiva completamente un elemento y evita que el usuario interactúe con él, mientras que `readonly` permite que el usuario vea el contenido, pero no le permite editarlo directamente. Ambas propiedades son útiles en diferentes situaciones según los requisitos específicos de interactividad en una página web.