Si queres poner un script en el html se pone al final de body

Si lo haces en un archivo .js lo llamas al final del body así:

<script src="js/01-app.js"></script>

Const en un objeto o arreglo si se puede modificar

**Propmt:**

Sale una alerta que te deja poner un texto y eso lo podes guardar en una variable:

const nombre= prompt('Cual es tu nombre?');

**QuerySelector:**

Seleccionar un elemento html por clase o id:

document.querySelector('.contenido');

**InnerHTML:**

Reemplaza contenido de html con lo que le indiques, acá reemplaza el contenido del div “.contenido” por Ema esta aprendiendo html

document.querySelector('.contenido').innerHTML = `${nombre} ESTA APRENDIENDO JAVASCRIPT`;

Para llamar las variables dentro de el innerHTML se hace con ${nombreVariable}

**Console:**

Console.log();

Console.table();

Console.error();

Console.warn();

Console.clear() limpia consola

Console.time(‘Hola’) Si metes código entre este y el de abajo te muestra cuanto tiempo tardo en ejecutarse

Console.timeEnd(‘Hola’);

**ESLint:**

Instalar:

En la consola de Windows para instalar global es

Npm i -g eslint

después vas a la carpeta del proyecto y crear un archivo que sea

.eslintrc.json

Te sirve para que cumplas con las formas de escribir código que indiques

Si ponemos esto en ese archivo va a controlar si faltan puntos y coma en el “semi”, lo primero creo que indica la versión de edmascript

{

    "parserOptions": {

        "ecmaVersion": 6

    },

    "rules": {

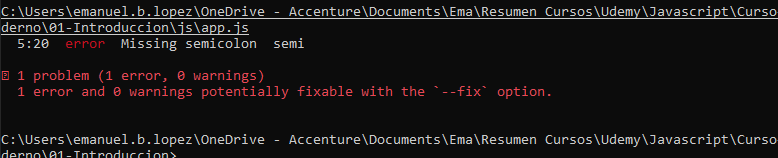
        "semi": ["error", "always"]

    }

}

Si te falta entonces en el js un punto y coma te das cuenta ejecutando en la consola, en la carpeta del proyecto. Eslint y la ruta del archivo .js

Eslint js/app.js



También te marca en rojo en el código js mientras lo vas escribiendo

Si tenes mucho código con errores lo podes arreglar desde la consola haciendo

Eslint js/app.js –fix

Entonces eso y ate arregla los errores en este caso pone los ;

**Variables:**

Si tenes varios archivos js vinculados a un html, no podes inicializar dos variables con el mismo nombre aunque estén en distintos archivos

Var: NO se usa casi, era para versiones anteriores

Var producto = “un producto”,

Edad= 26;

Let: Variable moderna

Const: son constantes no se pueden cambiar. Se tiene que inicializar con un valor, no se puede inicializar el valor después.

**Métodos extra:**

**Typeof:** Te indica el tipo de la variable

Typeof numero1 te puede devolver number o string o lo que sea

**Clases:**

Inician con letras mayúsculas

**Strings:**

Puede ir con comillas simples o dobles

Se puede hacer con:

String(‘Monitor de 24 pulgadas’)

O

New String(´Monitor de 24 pulgadas´)

Con este ultimo cada letra toma como una posición como si se creara un array o un objeto

Para que te tome las comillas dobles si estas envolviéndolo con comillas dobles podes escapar las comillas:

“Monitor de 24**\”**”; entonces te imprime monitor de 24”

**String métodos:**

**Length**: Te va a contar la cantidad de letras y espacios. NO usa paréntesis

console.log(producto.length);

**IndexOf:** Te busca en el string la palabra buscada y te devuelve la posición de la primer letra, devuelve -1 si no lo encuentra.

console.log(producto.indexOf('Monitor'));

**includes**: Tambien busca la palabra en el string , te devuelve true o false:

console.log(producto.includes('Monitor'));

**Concatenacion:**

Producto.concat(‘concateno esto a producto’);

Producto + “ Hola”

ES6: con comillas invertidas

console.log(`El producto ${producto} sale 20 pe`);

Otra es con coma:

console.log("El producto es " , producto , " sisi");

**trimStart():** Elimina los espacion en blanco del string

console.log(producto.trimStart());

**trimEnd():** Te elimina los espacios al final

console.log(producto.trimEnd());

Podes usar los dos métodos:

console.log(producto.trimStart().trimEnd());

o solo usar **trim()** que borra al principio y al final**:**

console.log(producto.trim());

**replace():** Para reemplazar una parte del string:

console.log(producto.replace('Pulgadas', 'pulg'));

**slice():** Para cortar el string 0 el inicio. EN este caso te va a dejar solo

console.log(producto.slice(0, 10));

console.log(producto.slice(8, 10));

**substring():** También corta pero, en este si le pones un numero mayor al principio lo da vuelta y lo corta bien igual, a diferencia del anterior. Acá cortaría del 8 al 10

console.log(producto.substring(10, 8));

**charAt():** Te corta la letra indicada

console.log(producto.charAt(0));

**repeat():** te repite el texto la cantidad de veces que le indiques

const texto= ' En promocion'.repeat(3);

**Split():** Divide el string por espacios por ejemplo

console.log(text.split(" "));

**toUppercase():** Para pasar a mayuscula

console.log(text.toUppercase());

**toLowerCase():** para pasar a minusula. Sirve para servir para pasar los mails a minúscula para que no haya conflictos en la base de datos

console.log(text.toLowerCase());

**Números:**

Let numero = 20;

Let numero1= new Number(80); este tec rea un objeto con el numero 80

**Objeto Math:**

Math.PI : te trae numero pi

Math.floor(2.9); redondea hacia abajo

Math.round(2.8) redondea a 3

Math.round(2.3) redondea a 2

Math.ceil(2.1) redondea hacia arriba

Math.sqrt(144); raiz cuadrada de 144

Math.abs(-500); ignora el negativo y te trae el valor absoluto sin el menos

Math.pow(8,3); es 8 a la potencia de 3

Math.min(2,3,4,1); te trae el valor mínimo en este caso 1

Math.max(2,3,5,7); te traeira el

Math.random(); numero aleatorio

Puntaje++

++puntaje primero agrega uno y después muestra puntaje

Puntaje—

--puntaje

**Métodos números:**

**Number.parseInt():** pasa de string a entero

Number.parseInt(“20”):

**Number.parseFloat():** conviente números flotantes

Number.parseFloat(“20.2”);

**Number.isInteger():** Controla si el numero es un entero (int)

Number..isInteger(20);

**Objetos:**

Un objeto agrupa todo en una sola variable.

Object literal:

const producto = {

    nombre: "Monitor 20 pulgadas", //propiedad

    precio: 300,

    disponible: true

}

Para acceder a los valores de un objeto:

Si queres acceder al nombre se puede hacer con:

console.log(producto.nombre);

o

console.log(producto['nombre']);

**Agregar y quitar propiedades al objeto:**

Para agregar le podes poner punto e inventar el nombre de la nueva propiedad, después le asignas el valor.

producto.imagen = "imagen.jpg";

Para borrar usas **delete** después el objeto.propiedad

delete producto.disponible;

**para guardar en una variable una propiedad con ES6 podes hacer:**

const nombre = producto.nombre;

//con ES6, creo que se crea la variable con nombre "nombre" de el objeto producto

const {nombre} = producto;

const {precio} = producto

console.log(nombre);

console.log(precio);

Se puede simplificar así:

const {nombre, precio} = producto;

console.log(nombre);

console.log(precio);

**Agregar objetos dentro de otro objeto:**

const producto = {

    nombre: "Monitor 20 pulgadas", //propiedad

    precio: 300,

    disponible: true,

    informacion:{

        peso: '1kg',

        medida: '1km'

    }

}

Llamada

console.log(producto.informacion.medida);

extraer:

const {nombre, informacion:{ fabricacion: {pais} }} = producto;

console.log(pais);

// Si queres extraer tambien fabricacion habria que ponerlo antes con una coma:

const {nombre, informacion:{fabricacion, fabricacion: {pais} }} = producto.informacion.fabricacion;

console.log(fabricacion);

Yo probe de esta forma y también funcionó:

const {pais} = producto.informacion.fabricacion;

console.log(pais);

**Modificar atributos del objeto:**

Podes modificarlos aunque el objeto este definido como const

**Métodos de objetos:**

**“use strict”:**

Para que un objeto se comporte como una constante y que no se puedan modificar los datos. Esto evalua el código de javascript y no permite malas practicas. Este código se pone arriba de todo

"use strict";

Esto te habilita algunos objet methods.

**Object.freeze():**

Te permite frizar un objeto y que no se pueda modificar atributos ni agregar o quitar:

Object.freeze(producto);

**Object.seal();**

La diferencia con freeze es que no se pueden agregar ni quitar propiedades pero si se puede modificar lo que existe

Object.seal(producto);

**Object.isSealed():**  Para saber si un objeto esta sellado

console.log(Object.isSealed(producto));

**Object.assign();** Te sirve para guardar en una variable la combinación de los atributos de dos objetos. Si tenes el objeto productos y medidas cada uno con sus atributos, los juntas creando una const así:

const resultado = Object.assign(producto, medidas);

Otra forma de hacerlo es:

const resultado2 = {...producto, ...medidas}; //los 3 puntos significa que copia

**Object constructor:**

Es otra forma de crear objetos con un constructor: Creamos una función en donde le pasamos paramentros de nombre y precio y el atributo disponible lo definimos directamente en true. Después con esto podemos crear varios productos:

function Producto(nombre, precio){

    this.nombre=nombre;

    this.precio=precio;

    this.disponible=true;

}

const producto2= new Producto('Monitor de 24 pulgadas', 500);

console.log(producto2)

const producto3= new Producto('Television', 1500);

console.log(producto3);

**Object.keys():** Con este método vas a poder traer el nombre de los atributos que tenga el objeto, x ej nombre, precio y disponible

console.log(Object.keys(producto));

Se usa también para saber si el objeto tiene o no tiene información

**Object.values();** Te retorna los valores de los atributos

console.log(Object.values(producto));

**Object.entries();** te retorna tanto el nombre de los atributos como sus valores

console.log(Object.entries(producto));

**THIS:**

Con this haces referencia a las propiedades del mismo objeto

En este caso al ponerle el this a nombre y precio los busca dentro del mismo objeto no por fuera

const producto = {

    nombre: "Monitor 20 pulgadas", //propiedad

    precio: 300,

    disponible: true,

    mostrarInfo: function(){

        console.log(`El producto ${this.nombre} tiene un precio de ${this.precio}`);

    }

}

**Arrays:**

**Creación de arrays;**

const numeros = [10, 20, 30];

o

const meses = new Array('Enero', 'Feberero', 'Marzo');

Podes mezclar tipos de datos y hasta poner objetos:

const deTodo = ["Hola", 10, true, null, {nombre: 'Juan', trabajo:'programador'}];

**Acceder a elementos de un array:**

Con el index:

console.log(numeros[2]);

**Recorrer array:**

for(let i=0;i<meses.length;i++){

    console.log(meses[i]);

}

**Modificar elementos del array:**

A pesar de que un array este inicializado con const se pueden modificar los elementos haciendo

meses[0] = 'nuevoMes';

**Agregar elementos:**

**Push():**

meses.push('Julio');

Acá le agregamos un objeto de un producto al array:

const carrito = [];

const producto = {

    nombre: 'Monitor de 32 pulgadas',

    precio: 500,

}

carrito.push(producto);

**Unshift();** Para llevar un producto al principio de un array:

carrito.unshift(producto3);

Otra forma de hacerlo es así:

let resultado;

resultado = [...carrito, producto];

resultado = [...resultado, producto2];

resultado = [producto3, ...resultado];

**pop():** Para eliminar el ultimo elemento de un array

carrito.pop();

**shifht();** Eliminar elemento del inicio del array

carrito.shift();

**splice();** Para eliminar de cualquier parte del array, primer se pone el numero de la posición donde va a empezar a cortar (Empezando desde 0) y segundo es cuantos elementos se quieren eliminar. En este caso eliminamos solo el elemento de la posición 1

carrito.splice(1,1);

**Destructuring de arrays:**

En este caso los nombres que indiques en [] se van a ir asignando por orden

const numeros = [10,20,30,40,50];

const [primero, segundo, tercero] = numeros;

console.log(tercero);

primero = 10;

segundo = 20;

tercero = 30;

etc.

Si necesitas sacar solamente el 3er valor x ej. Vas a hacer así:

const numeros = [10,20,30,40,50];

const [ , , tercero] = numeros;

console.log(tercero);

En este caso tercero va a contener 30,40 y 50

const numeros = [10,20,30,40,50];

const [ primero, segundo, ...tercero] = numeros;

console.log(tercero);

**ForEach:**

En este caso:

const carrito = [

    {nombre: 'Monitor de 20 pulgadas', precio: 500},

    {nombre: 'Television', precio: 100},

    {nombre: 'Tablet', precio: 200},

    {nombre: 'Audifonos', precio: 300},

    {nombre: 'Teclado', precio: 400},

    {nombre: 'Celular', precio: 700},

]

Si haces un for común para acceder a los elementos del array te va a mostrar cada uno de los objetos.

Entonces si quisieras acceder a cada uno de el nombre y precio tendrías que hacer algo como:

for(let i = 0; i < carrito.length; i++){

    console.log(`${carrito[i].nombre} - Precio: ${carrito[i].precio}`);

}

Pero con un forEach podes hacer asi:

carrito.forEach( function(producto){

    console.log(`${producto.nombre} - Precio: ${producto.precio}`);

})

Producto va a ir almacenando cada uno de los objetos del array, e imprimiendo su nombre y su precio

**Map():**

La diferencia con foreach, es que map podes crear una variable con un nuevo array con los elementos que recorre con el foreach no pode hacer eso, solo los recorre.

const nuevoArreglo = carrito.map( function(producto){

    return `${producto.nombre} - Precio: ${producto.precio}`;

})

**Funciones:**

**Declaración de función:** En este caso se puede llamar a la función antes de inicializarlo ósea el sumar() antes de function sumar().

function  sumar(){

    console.log(2 + 2);

}

sumar();

**Expresión de función:** En este caso NO se puede llamar a la función antes de inicializarlo ósea el sumar2() antes de const sumar2 = function(){}

const sumar2 = function(){

    console.log(3 + 3);

}

sumar2();

**-----------------**

javascript escanea dos veces el documento, primero ve todas las variables y funciones y en la segunda se ejecuta. Esto se conoce como hoisting

**funciones nativas:**

Si vos escribis **window** en la consola del browser y le das enter te aparecen todas las funciones

**Diferencia entre función y método:**

En una función cuando se la llama primero va el nombre de la función y entre paréntesis la variable.

En un método primero va el nombre de la variable después un punto y el nombre del método

const numero1 = 20;

const numero2 = '20';

console.log(parseInt(numero2)); // funcion

console.log(numero1.toString()) // metodo

**Parámetros y argumentos:**

function  sumar(a,b){  // a y b son parametros

    console.log(a + b);

}

sumar(2, 3); // argumentos

function saludar(nombre, apellido){

    console.log(`Hola ${nombre} ${apellido}`);

}

saludar('Ema', 'Lopez');

**Parámetros por default:**

En este caso si no le pasas los parámetros podes definir unos por default indicándolo con “igual” al lado de los parámetros, si le mandas argumentos van a tomarse los que se envíen.

function saludar(nombre = 'No tiene nombre', apellido = 'No tiene apellido'){

    console.log(`Hola ${nombre} ${apellido}`);

}

// saludar('Ema', 'Lopez');

saludar();

**Como se comunican las funciones?**

iniciarApp();

function iniciarApp(){

    console.log("Iniciar app");

    segundaFuncion();

}

function segundaFuncion(){

    console.log("Desde la segunda funcion");

    usuarioAutenticado("pablo");

}

function usuarioAutenticado(usuario){

    console.log("Autenticando usuario");

    console.log(`Uusario autenticado: ${usuario}`);

}

**Funciones que retornan valores:**

function sumar(a,b){

    return a + b;

}

const resultado= sumar(4,5);

console.log(resultado);

let total = 0;

function agregarcarrito(precio){

    return total += precio;

}

function calcularImpuesto(total){

    return total \* 1.15;

}

total = agregarcarrito(300);

total = agregarcarrito(100);

total = agregarcarrito(600);

const totalPagar = calcularImpuesto(total);

console.log("El total a pagar es: " + totalPagar);

**Añadir funciones a un objeto:**

Al tener una función dentro de un objeto se lo conoce como método de propiedad

const reproductor = {

    reproducir: function(id){

        console.log(`Reproduciendo cancion... numero ${id}`);

    },

    pausar: function(){

        console.log('Pausando...')

    },

    crearPlaylist: function(nombre){

        console.log(`Creando playlist ${nombre}`)

    },

    reproducirPlaylist: function(nombre){

        console.log(`Reproduciendo Playlist ${nombre}`)

    }

}

reproductor.reproducir(30);

reproductor.reproducir(20);

reproductor.pausar();

reproductor.borrar = function(id){

    console.log(`Borrando cancion... numero: ${id}`);

}

reproductor.borrar(25);

reproductor.crearPlaylist('Para trabajar');

reproductor.reproducirPlaylist('Para trabajar');

**Arrow functions:**

Para hacer una función flecha se hace así:

const aprendiendo = function(){

    console.log('Aprendiendo javascript');

}

const aprendiendo2 = () =>{

    console.log('Aprendiendo javascript');

}

aprendiendo2();

Si tenes una sola instrucción podes sacar las llaves y hace solo el return, ósea que en aprendiendo2 va a retornar “aprendiendo javascript” cuando llames aprendiendo2()

const aprendiendo2 = () => 'Aprendiendo javascript';

console.log(aprendiendo2());

**Arrow functions con parámetros:**

const aprendiendo = function(tecnologia){

    console.log(`Aprendiendo ${tecnologia}`);

}

aprendiendo('Javascript');

const aprendiendo2 = (tecnologia) => `Aprendiendo ${tecnologia}`;

console.log(aprendiendo2('Javascript'));

**map y forEach con arrow functions:**

carrito.forEach( (producto) => {

    console.log(`${producto.nombre} - Precio: ${producto.precio}`);

})

const nuevoArreglo = carrito.map( (producto) => {

    return `${producto.nombre} - Precio: ${producto.precio}`;

})

**O**

Al ser un solo parámetro se le pueden quitar los paréntesis y al tener una sola línea se pueden quitar las llaves

const nuevoArreglo = carrito.map( producto => `${producto.nombre} - Precio: ${producto.precio}`)

carrito.forEach( producto => `${producto.nombre} - Precio: ${producto.precio}`)

mas ejemplos:

const reproductor = {

    reproducir: id => console.log(`Reproduciendo cancion... numero ${id}`),

    pausar: () => console.log('Pausando...'),

    crearPlaylist: nombre => console.log(`Creando playlist ${nombre}`),

    reproducirPlaylist: nombre => console.log(`Reproduciendo Playlist ${nombre}`)

}

**Setters y getters:**

const reproductor = {

    cancion: '',

    set nuevaCancion(cancion){

        this.cancion = cancion;

        console.log(`Añadiendo ${cancion}`);

    },

    get obtenerCancion(){

        console.log(`${this.cancion}`)

    }

}

reproductor.nuevaCancion = 'Enter Sandman';

reproductor.obtenerCancion;

****

**Estructuras:**

switch(metodoPago){

    case 'Efectivo':

        consolelog("Pagaste con efectivo");

        break;

    case 'Cheque':

        console.log("Pagaste con cheque");

        break;

    default:

        console.log("Metodo de pago no soportado o no seleccionaste");

        break;

}

**If anidado ternario:**



const autenticado= true;

const puedePagar = true;

console.log(autenticado ? puedePagar ? 'Si esta autenticado y puede pagar' : 'si autenticado pero no puede pagar' : 'No no esta autenticado');

Si esta autenticado entra por el if primero y ahí consulta también si puede pagar, si puede pagar es true ‘Si esta autenticado y puede pagar’ si puede pagar es false ‘si esta autenticado pero no puede pagar. Si autenticado es false ‘No esta autenticado’

**Iteradores:**

**Break y continue en fors:**

Con el break termina el for y no se vuelve a ejecutar

for(let i=0; i<=10; i++){

    if(i === 5){

        console.log("Este es el 5");

        break;

    }

    console.log(i);

}

Con el continue no detiene la ejecución del forloop en vez de seguir con las líneas de abajo dentro del for lo vuelve a correr. E este caso va a mostrar:

for(let i=0; i<=10; i++){

    if(i === 5){

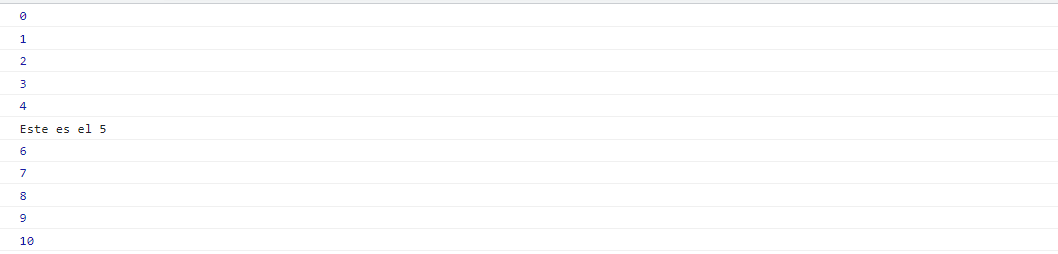
        console.log("Este es el 5");

        continue;

    }

    console.log(i);

}



**Fizz buzz:**

for(let i=0;i<=100;i++){

    if(i % 15 === 0){

        console.log(`${i} fizz buzz`);

    }else if(i % 5 === 0){

        console.log(`${i} buzz`);

    }else if(i % 3 === 0){

        console.log(`${i} fizz`);

    }

}

forEAch:

Con índice (empieza desde 0)

const pendientes = ['Comer', 'Dormir', 'Estudiar', 'Correr'];

pendientes.forEach((pendiente, index) => {

    console.log(`${index}: ${pendiente}`);

})

const carrito = [

    {nombre: 'Monitor de 20 pulgadas', precio: 500},

    {nombre: 'Television', precio: 100},

    {nombre: 'Tablet', precio: 200},

    {nombre: 'Audifonos', precio: 300},

    {nombre: 'Teclado', precio: 400},

    {nombre: 'Celular', precio: 700},

]

carrito.forEach(producto => console.log(producto.nombre));

El map es lo mismo solo que crea un nuevo array.

**For of:**

Let nombreRandom of array. Este itera sobre arreglos

const pendientes = ['Comer', 'Dormir', 'Estudiar', 'Correr'];

const carrito = [

    {nombre: 'Monitor de 20 pulgadas', precio: 500},

    {nombre: 'Television', precio: 100},

    {nombre: 'Tablet', precio: 200},

    {nombre: 'Audifonos', precio: 300},

    {nombre: 'Teclado', precio: 400},

    {nombre: 'Celular', precio: 700},

]

for(let pendiente of pendientes){

    console.log(pendiente);

}

for(let producto of carrito){

    console.log(producto.nombre);

}

**For in:**

Este itera sobre objetos es mas conveniente parece

For(let propiedad in automovil){

Console.log(`${automovil}[propiedad]`;

}

Sino una forma mas fácil de recorrer en objetos es así, en donde llave va a guardar el nombre del atributo y valor el valor de cada uno de los atributos

const automovil = {

    modelo: 'Camaro',

    year: 1969,

    motor: 6.0

}

for(let [llave, valor] of Object.entries(automovil)){

    console.log(valor);

}

**Array methods:**

**Some():**

Comprobar si un valor existe en un arreglo como este:

const meses = ['Enero', 'Febrero', 'Marzo', 'Abril', 'Mayo', 'Junio', 'Julio'];

const resultado = meses.includes('Enero'); // devuelve bool

console.log(resultado);

En un arreglo de objetos no va a funcionar y hay que usar some():

const carrito = [

    { nombre: 'Monitor 27 Pulgadas', precio: 500 },

    { nombre: 'Televisión', precio: 100 },

    { nombre: 'Tablet', precio: 200 },

    { nombre: 'Audifonos', precio: 300 },

    { nombre: 'Teclado', precio: 400 },

    { nombre: 'Celular', precio: 700 },

]

const existe= carrito.some((producto) =>{

    return producto.nombre === 'Celular';

})

Aunque también este se puede usar en arreglos comunes:

const existe2= meses.some((mes) =>{

    mes === 'Celular';

})

**findIndex():**

Encontrar el índice de un elmento del array, devuelve -1 si no encuentra

const indice = meses.findIndex(mes => mes === 'Abril')

const indice2 = carrito.findIndex((producto) =>{

    return producto.nombre === 'Tablet';

});

**Reduce():**

Si queres saber el total a apagar x ej, el 0 del final es el valor inicial de total.

let total=0;

let resultado = carrito.reduce((total, producto)=>{

    total + producto.precio, 0;

})

**Filter():**

Crea un nuevo array basado en el parámetro que es evaluado: Acá por ejemplo filtramos por los productos con precio mayor a 400

let resultado;

resultado = carrito.filter(producto => preducto.precio > 400);

Si queres que te traiga todos menos uno podes hacer:

resultado = carrito.filter(producto => producto.nombre !== 'Audifonos');

**find():**

Crea un nuevo arreglo según la condición:

Se extrae uno de los arrays del objeto:

Así seria con foreach

let resultado = '';

carrito.forEach((producto, index)=> {

    if(producto.nombre == 'Tablet'){

        resultado = carrito[index];

    }

});

console.log(resultado);

Así de la otra forma:

const resultado2 = carrito.find(producto => producto.nombre === 'Tablet');

OJO QUE ESTE VA A TRAER UNO SOLO, si se quiere traer varios hacerlo con find();

**Every():**

Con este todos los elementos de un arreglo deben cumplir la condición para que nos de true

En este caso todos los elementos tienen que ser menores de 500 pesos para que devuelva true, si uno no cumple ya devuelve false

const resultado = carrito.every(producto => producto.precio < 500);

**Concat():**

Une o concatena 2 arreglos:

const meses = ['Enero', 'Febrero', 'Marzo', 'Abril', 'Mayo', 'Junio', 'Julio'];

const meses2 = ['Agosto', 'Septiembre'];

const meses3 = meses.concat(meses2);

Podes concatenar mas de 1 con una coma

const meses = ['Enero', 'Febrero', 'Marzo', 'Abril', 'Mayo', 'Junio', 'Julio'];

const meses2 = ['Agosto', 'Septiembre'];

const meses3 = ['Octubre', 'Noviembre'];

const meses4 = meses.concat(meses2, meses3, ‘Otro mes’);

También se puede hacer con spread operator:

const resultado2 = [...meses, ...meses2, ...meses3];

**Spread Operator:**

const meses = ['Enero', 'Febrero', 'Marzo', 'Abril', 'Mayo', 'Junio', 'Julio'];

const meses2 = [...meses, 'Agosto'];

console.log(meses2);

Acá no modifica el arreglo original, crea uno nuevo agregándole agosto. Si usaras .push() para agregar modifica el arreglo original.

En la programación funcional se busca no modificar el array original.

Para objetos igual:

const carrito = [

    { nombre: 'Monitor 27 Pulgadas', precio: 500 },

    { nombre: 'Televisión', precio: 100 },

    { nombre: 'Tablet', precio: 200 },

    { nombre: 'Audifonos', precio: 300 },

    { nombre: 'Teclado', precio: 400 },

    { nombre: 'Celular', precio: 700 },

]

const producto = {nombre: 'Disco Duro', precio: 300};

const carrito2 = [...carrito, producto];

**DOM:**

Que es?

Si tenes un archivo html , javascript se refiere a el como Document, ese document es el objeto principal del DOM. Y también tenemos el root que es el html. De ahí se divide en 2 el head y el body. El head tiene una etiqueta title y dentro de este un texto. En el body podes tener diferentes elementos.

Es el código html y podes acceder a todo el contenido seleccionar, editar, etc..

**Document.:**

Podes seleccionar todo desde el document aunque no sea lo que mas se usa

let elemento;

elemento = document;

elemento = document.all;

elemento = document.head;

elemento = document.body;

elemento = document.domain;

elemento = document.forms;

elemento = document.forms[0]; // con el indice seleccionas cual link es con el que queres trabajar

elemento = document.forms[0].id; //retorna id

elemento = document.forms[0].method;

elemento = document.forms[0].classList; // retorna su clase como TokenList

elemento = document.forms[0].action; // retorna url

elemento = document.links;

elemento = document.links[4]; // con el indice seleccionas cual link es con el que queres trabajar

elemento = document.links[4].className; // retorna su clase como string

elemento = document.images;

elemento = document.scripts;

**getElementsByClassName:**

const header = document.getElementsByClassName('header');

respetar mayusculas y minusculas. Si la clase no existe te retorna como un array vacio.

**getElementById:**

const formulario = document.getElementById('formulario');

Si tenes mas de un id igual te retorna el primero que encuentre.

**querySelector:**

Te trae de las clases el primero que encuentra

const card = document.querySelector('.card');

const info = document.querySelector('.premium .info');

const segundoCard = document.querySelector('.hospedaje .card:nth-child(2)');

const form = document.querySelector('#formulario');

**querySelectorAll:**

te retorna todos los elementos que cumplan la condición

const cards = document.querySelectorAll('.card');

**Acceder al texto:**

const encabezado = document.querySelector('.contenido-hero h1');

console.log(encabezado.innerText); //no me lo recomendaba eleditor si esta hidden no lo encuentra

console.log(encabezado.textContent); // trae el texto, respeta espacios, si esta hidden si lo encuentra

console.log(encabezado.innerHTML); // trae el html si tiene span dentro lo trae

Podes hacer sino:

const encabezado = document.querySelector('.contenido-hero h1').textContent;

Así cambias el texto:

document.querySelector('.contenido-hero h1').textContent = 'Nuevo texto';

**Seleccionar imagen y cambiarla:**

document.querySelector('.card img').src = 'img/imagen2.jpg';

**Cambiar CSS:**

En javascript, cuando vos tenes en CSS por ejemplo justify-content se pone justifyContent. Se saca el guion y se hace camelCase.

Cambiamos el color de fondo de un h1

document.querySelector('h1').style.backgroundColor = 'red';

document.querySelector('h1').style.height = '180px';

siempre con comillas

**Agregar o quitar clases:**

Agregar:

Le agrega a card ‘nuevaClase’

document.querySelector('.card').classList.add('nuevaClase');

document.querySelector('.card').classList.add('nuevaClase', 'segundaClase');

y ahí se agregan 2

**Para borrar una clase:**

Borramos la clase card

document.querySelector('.card').classList.remove('card');

**ChildNodes y children:**

Te trae los elementos todos los nodos. Acá los espacios en blanco son considerados elementos como los saltos de línea en los que apretas enter para hacer otra etiqueta por ej.

const navegacion = document.querySelector('.navegacion').childNodes;

Con children no te los va a tomar por lo que va a ser mas efectivo y podes seleccionar similar a un array cual de los elementos que trae es el que queres seleccionar:

const navegacion = document.querySelector('.navegacion').children[0];

si queres retornar la etiqueta de ese se puede hacer así te va a devolver ‘a’ si es un enlace x ej.

const navegacion = document.querySelector('.navegacion').children[0].nodeName;

podes saber el tipo de nodo así, te devuelve números que indican que tipo de elemento es.

const navegacion = document.querySelector('.navegacion').children[0].nodeType;

**Traversing de hijo a padre:**

Para acceder al elemento padre se usa **parentElement**

const card = document.querySelector('.card');

console.log(card.parentElement)

Si queres seguir subiendo, le seguis agregando:

const card = document.querySelector('.card');

console.log(card.parentElement.parentElement.parentElement)

**Acceder a hermanos:**

Hermano siguiente:

const card = document.querySelector('.card');

console.log(card.nextElementSibling)

console.log(card.nextElementSibling.nextElementSibling)

Hermano anterior:

const ultimaCard = document.querySelector('.card:nth-child(4)');

console.log(ultimaCard.previousElementSibling);

**Primer elemento hijo:**

const navegacion = document.querySelector('.navegacion');

console.log(navegacion.firstElementChild);

**Ultimo element hijo:**

const navegacion = document.querySelector('.navegacion');

console.log(navegacion.lastElementChild);

**Eliminar elementos del DOM:**

Por si mismo

const primerEnlace = document.querySelector('a');

primerEnlace.remove();

Por referencia del padre:

const navegacion = document.querySelector('.navegacion');

navegacion.removeChild(navegacion.children[2]);

**Generar HTML con Javascript:**

const enlace = document.createElement('a');

enlace.textContent = 'Nuevo enlace';

//agregamos el href

enlace.href = '/nuevo-enlace';

enlace.target = "\_blank";

//seleccionar donde se va a agregar

const navegacion = document.querySelector('.navegacion');

//le agregamos el enlace

navegacion.appendChild(enlace);

Podes insertarlo antes de la posición numero 1 de los hijos de navegación así:

navegacion.insertBefore(enlace, navegacion.children[1]);

CREAMOS UNA CARD CON JS:

const parrafo1 = document.createElement('p');

const parrafo2 = document.createElement('p');

const parrafo3 = document.createElement('p');

parrafo1.textContent = 'Concierto';

parrafo1.classList.add('categoria', 'concierto');

parrafo2.textContent = 'Concierto de rock';

parrafo2.classList.add('titulo');

parrafo3.textContent = '$800 x persona';

parrafo3.classList.add('precio');

//creamos div con la clase de info

const info = document.createElement('div');

info.classList.add('info');

info.appendChild(parrafo1);

info.appendChild(parrafo2);

info.appendChild(parrafo3);

//creamos la imagen

const imagen = document.createElement('img');

imagen.src = 'img/hacer2.jpg';

//creamos el card

const card = document.createElement('div');

card.classList.add('card');

//asignamos la imagen

card.appendChild(imagen);

//asignar info

card.appendChild(info);

console.log(card)

//insertar todo en el html

const contenedor = document.querySelector('.hacer .contenedor-cards');

contenedor.appendChild(card);

**Ejercicio footer:**

Función que pregunta si el footer tiene la clase activo se la saca y sino se la agrega para ocultar o mostrar

function mostrarOcultarFooter(){

    if(footer.classList.contains('activo')){

        footer.classList.remove('activo');

    }else{

        footer.classList.add('activo');

    }

}

**Eventos:**

**Click:**

Acá cuando le da click a el botón seleccionado ejecuta esa función:

const btnFlotante = document.querySelector('.btn-flotante');

btnFlotante.addEventListener('click', mostrarOcultarFooter);

function mostrarOcultarFooter(){

    console.log('Diste click');

}

**DOMContentLoaded:**

Se ejecuta una vez que es descargado todo el html:

document.addEventListener('DOMContentLoaded', () =>{

    console.log(‘se cargo el html’);

})

**Mouseenter:**

Se ejecuta cuando haces hover:

const nav = document.querySelector('.navegacion');

nav.addEventListener('mouseenter', () => {

    console.log('Entrando a la navegacion');

})

**Mouseout:**

Cuando dejas de hacer hover

const nav = document.querySelector('.navegacion');

nav.addEventListener('mouseout', () => {

    console.log('saliendo de la navegacion');

})

**Mousedown:**

Cuando apretas en el elemento similar a click

nav.addEventListener('mousedown', () => {

    console.log('presionando navegacion');

})

**Mouseup:**

Cuando soltas el click

nav.addEventListener('mouseup', () => {

    console.log('presionando navegacion');

})

**Dblclick:**

Cuando das doble click

nav.addEventListener('dblclick', () => {

    console.log('doble click navegacion');

})

**Eventos en inputs:**

**Keydown:**

Cuando estas presionando una tecla

const busqueda = document.querySelector('.busqueda');

busqueda.addEventListener('keydown', () => {

    console.log('escribiendo...');

})

**Keyup:**

Cuando dejas de presionar una tecla

const busqueda = document.querySelector('.busqueda');

busqueda.addEventListener('keyup', () => {

    console.log(dejando de escribir...');

})

**Blur:**

Tenes que entrar al input, cuando cliqueas en otro lado se ejecuta:

const busqueda = document.querySelector('.busqueda');

busqueda.addEventListener('blur', () => {

    console.log('campo invalido ...');

})

**Copy:**

Cuando escribiste un texto en el input y lo copias, cuando lo copias se ejecuta

const busqueda = document.querySelector('.busqueda');

busqueda.addEventListener('copy', () => {

    console.log('No copiar!');

})

**Paste:**

Cuando pegas algo

const busqueda = document.querySelector('.busqueda');

busqueda.addEventListener('paste', () => {

    console.log(‘pegaste!');

})

**Cut:**

Cuando cortas

const busqueda = document.querySelector('.busqueda');

busqueda.addEventListener('cut', () => {

    console.log('Cortaste!');

})

**Input:**

Cuando se realiza cualquiera de los anteriores menos el blur

Acá tomamos con e el evento de input y con e.target.value va guardando todo lo que el usuario va escribiendo, puede servir para búsquedas:

const busqueda = document.querySelector('.busqueda');

busqueda.addEventListener('input', (e) => {

    console.log(e.target.value);

})

**Eventos de formulario:**

**Submit:**

Cuando se da click en el botón de submit de enviar, se le pasa el evento con e.

Por default el submit hace un POST via la url que se le indique, entonces si le hacemos el e.preventDefault(); prevenimos la acción por defecto, como puede ser enviar el formulario o en un link entrar al link.

formulario.addEventListener('submit', (e) => {

    e.preventDefault();

})

O

formulario.addEventListener('submit', validarFormulario);

function validarFormulario(e){

    e.preventDefault();

}

**Eventos de Scroll:**

Se ejecuta cada vez que se hace scroll

window.addEventListener('scroll', () => {

    console.log('scrolling');

})

Para saber la cantidad de scroll que haces en px

window.addEventListener('scroll', () => {

    const scrollPX = window.scrollY;

    console.log(scrollPX);

})

Para detectar cuando un elemento esta visible:

Seleccionamos el elemento, obtenemos la ubicación del mismo, vemos cual es el top o el bottom con la consola debugeando, podes hacer un console.log(ubicación) y fijarte a que top o bottom se tiene que ver

window.addEventListener('scroll', () => {

    const premium = document.querySelector('.premium');

    const ubicacion = premium.getBoundingClientRect();

    if(ubicacion.top < 784){

        console.log('El elemento ya se ve');

    }else{

        console.log('Aun no se ve');

    }

})

**Event bubbling:**

Cuando presionas en un evento pero el evento se propaga por otros lugares dando resultados que no son los esperados.

En este caso info esta dentro de card, y cuando seleccionar info se propaga y también se ejecuta el click de card. Y si apretas en titulo se dispara ese + el de info y card

const carDiv = document.querySelector('.card');

const infoDiv = document.querySelector('.info');

const tituloDiv = document.querySelector('.titulo');

carDiv.addEventListener('click', () =>{

    console.log('click en card');

});

infoDiv.addEventListener('click', () =>{

    console.log('click en info');

});

tituloDiv.addEventListener('click', () =>{

    console.log('click en titulo');

});

Para solucionarlo usamos el stopPropagation()

carDiv.addEventListener('click', (e) =>{

    e.stopPropagation();

    console.log('click en card');

});

infoDiv.addEventListener('click', (e) =>{

    e.stopPropagation();

    console.log('click en info');

});

tituloDiv.addEventListener('click', (e) =>{

    e.stopPropagation();

    console.log('click en titulo');

});

**Delegation de evento bubbling:**

Con e.target podes identificar a que le estas dando click dentro de l div que se selecciono

carDiv.addEventListener('click', (e) => {

    console.log(e.target);

})

Se consulta si donde se dio click tiene la clase

const carDiv = document.querySelector('.card');

carDiv.addEventListener('click', (e) => {

    if(e.target.classList.contains('titulo')){

        console.log('Diste click en titulo')

    }

    if(e.target.classList.contains('precio')){

        console.log('Diste click en precio')

    }

    if(e.target.classList.contains('card')){

        console.log('Diste click en card')

    }

})

**Prevenir evento bubbling con un método:**

function nuevaFuncion(a){

    console.log('llamada id ' +  a);

}

parrafo3.onclick = function(){

    nuevaFuncion(1);

};

**Con e.target sabremos cual es el elemento que se esta clickeando.**

**.length solo lo podes hacer con queryselectorAll no con querySelector común**

**Validacion de mails:**

**<https://emailregex.com/>**

    if(e.target.type === 'email'){

        const er = /^(([^<>()\[\]\\.,;:\s@"]+(\.[^<>()\[\]\\.,;:\s@"]+)\*)|(".+"))@((\[[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}])|(([a-zA-Z\-0-9]+\.)+[a-zA-Z]{2,}))$/;

        if(er.test(e.target.value)){

            console.log('El email es valido');

        }else{

            console.log('email no valido')

        }

    }

**SetTimeOut:**

    setTimeout(() =>{

        console.log('despues 3 seg');

    }, 3000);

**SetInterval:**

Cada 3 segundos se ejecuta

    setInterval(() =>{

        console.log('cada 3 seg');

    }, 3000);

**Resetear formulario:**

formulario.reset();

**Date:**

new Date().getFullYear();

Crea un number con el año actual 2022

**Local Storage:**

**Agregar datos:**

Con el local storage los datos se mantienen aunque cierres la pagina apagues la compu o lo que sea. Solo almacenan strings

localStorage.setItem('nombre', 'Ema'); // se pone una llave (nombre) que le asignas vos y el valor

Se puede poner u objeto pero se tiene que convertir a string primero así:

const producto = {

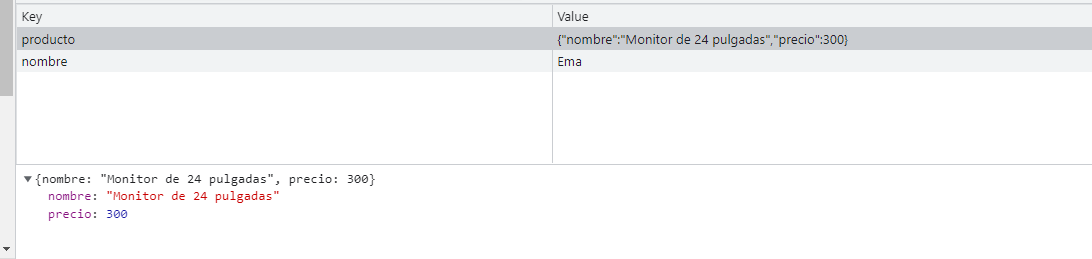
    nombre: "Monitor de 24 pulgadas",

    precio: 300

}

const productoString = JSON.stringify(producto); //convierte el objeto en un string

localStorage.setItem('producto', productoString);



Lo mismo para los arrays:

const meses = ['Enero', 'Febero', 'Marzo'];

const mesesString = JSON.stringify( meses );

localStorage.setItem('meses', mesesString);

**Obtener datos:**

const nombre = localStorage.getItem('nombre');

Si queres traer un objeto te lo va a traer como string. Entonces hay que convertirlo de string a objeto:

const producto1 = localStorage.getItem('producto');

const objetoProducto = JSON.parse(producto1);

console.log(objetoProducto);

Lo mismo para arrays:

const meses1 = localStorage.getItem('meses');

const mesesArray = JSON.parse(meses1);

**Eliminar elementos del local Storage:**

localStorage.removeItem('nombre');

**Actualizar:**

const mesesArray2= JSON.parse(localStorage.getItem('meses'));

mesesArray2.push('Nuevo mes');

localStorage.setItem('meses', JSON.stringify(mesesArray2));

**Eliminar todo el local storage:**

localStorage.clear();

**Session Storage:**

Cuando se cierra el navegador se pierden los datos

sessionStorage.setItem('apellido', 'Lopez'); // se pone una llave (nombre) que le asignas vos y el valor

**Fechas:**

**Date():**

Te devuelve un objeto con día/ mes / año / hora / gmt / (de donde)

const diaHoy = new Date();

let valor;

valor = diaHoy;

console.log(valor);

Eso imprime esto:

Mon Jan 31 2022 09:54:50 GMT-0300 (Argentina Standard Time)

Se le puede pasar el que vos quieras con mes-dia-año

const diaHoy = new Date('1-5-2000');

o Tambien:

const diaHoy = new Date('January 5 2000');

**Otros:**

Tiene metodos con get que es para traer el valor y otros con set que son para setear valores

valor = diaHoy.getFullYear(); // retorna año actual

valor = diaHoy.getMonth(); // retorna mes actual inicia en 0 a enero!

valor = diaHoy.getMinutes(); // retorna los minutos acutales

valor = diaHoy.getHours(); // retorna la hora actual

valor = diaHoy.getTime(); // retorna los milisegundos desde el 1 de enero de 1970 hasta la fecha

valor = diaHoy.setFullYear('2010');

**Cambiar formato de date:**

valor = diaHoy.toLocaleString();

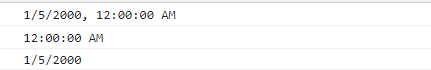
console.log(valor);

valor = diaHoy.toLocaleTimeString();

console.log(valor);

valor = diaHoy.toLocaleDateString();

console.log(valor);

****

**Momentjs.com:**

Es una librería

Scripts para el html:

    <script src="https://cdnjs.cloudflare.com/ajax/libs/moment.js/2.27.0/moment.min.js"></script>

    <script src="https://cdnjs.cloudflare.com/ajax/libs/moment.js/2.27.0/locale/es.min.js" integrity="sha512-tgY2qswcbQir80Vp67s5ZdbKikl99YmVXp3V/C4Acthk4gI29ONbQ+MR8B5tpESkNoa0N1P7HnSuzC6nOflrwA==" crossorigin="anonymous"></script>

Conviene ver la documentacion de la pagina para saber todas las variables

En este caso, le indicamos mes , día, año, horas:minutos:segundos , AM o PM. Y con locale le indicamos el idioma.

moment.locale('es');

console.log(moment().format('MMMM Do YYYY h:mm:ss a'))

**Prototipes:**

Todos los objetos tienen un prototype. Cuando abris un objeto en la consola te muestra el prototype y son funciones exclusivas del objeto

Para crear objetos lo podes hacer así:

const cliente = {

    nombre: 'Ema',

    saldo: 500

}

O para hacerlo mas dinámico, hay que hacer como object constructor entonces así podes definir varios objetos

function Cliente(nombre,saldo){

    this.nombre = nombre;

    this.saldo = saldo;

}

const juan = new Cliente('Juan', 500);

**Que problema solucionan los prototypes?**

Ponele que agregamos un objeto de empresa

function Empresa(nombre,saldo, categoria){

    this.nombre = nombre;

    this.saldo = saldo;

    this.categoria = categoria;

}

function formatearEmpresa(empresa){

    const {nombre, saldo, categoria} = empresa;

    return `El cliente ${nombre} tiene un saldo de ${saldo} y pertenece a la categoria ${categoria}`;

}

const CCJ = new Empresa('Codigo con Juan', 4000, 'Cursos en linea');

console.log(formatearEmpresa(CCJ));

y para saber cual función le pertenece a cada objeto sirve el prototype, entonces expandis el prototype y sabes cual función es de cada uno.

**Para agregar funciones al proto:**

function Cliente(nombre,saldo){

    this.nombre = nombre;

    this.saldo = saldo;

}

Cliente.prototype.tipoCliente = function (){ // tipocliente va a ser la funcion

    console.log('desde mi nuevo proto');

}

const pedro = new Cliente('Pedro', 6000);

console.log(pedro);

Entonces podes llamar a la función desde el objeto:

pedro.tipoCliente();

podes llamar a las propiedades del objeto con this. Dentro de la función del prototype

Cliente.prototype.tipoCliente = function (){ // tipocliente va a ser la funcion

    console.log(this.saldo);

}

Otro ejemplo:

Cliente.prototype.tipoCliente = function (){ // tipocliente va a ser la funcion

    let tipo;

    if(this.saldo > 10000){

        tipo= 'Gold';

    }else if (this.saldo > 5000){

        tipo = 'Platinum';

    }else{

        tipo = 'Normal';

    }

    return tipo;

}

**Heredar prototypes y constructores:**

En este caso se hereda de cliente las propiedades nombre y saldo con el método call

function Persona(nombre,saldo,telefono){

    Cliente.call(this, nombre, saldo)

    this.telefono = telefono;

}

Para heredar también las funciones se hace:

Persona.prototype = Object.create(Cliente.prototype);

Persona.prototype.constructor = Cliente;

Esto hay que hacerlo antes de instanciar el objeto

**POO:**

**Creación de clases:**

**Hay 2 formas de declararlas:**

Class declaration:

class Cliente {

}

Class Expression (esta forma casi no se usa):

const Cliente2 = class {

}

**Instanciar clase:**

const juan = new Cliente();

**Constructor de una clase:**

class Cliente {

    constructor(nombre,saldo){

        this.nombre = nombre;

        this.saldo = saldo;

    }

}

const juan = new Cliente('Juan', 400);

En class expression es igual

**Crear métodos y llamarlos con clases:**

Crea los métodos dentro de la clase se llama por el nombre de la clase, punto y nombre del método

class Cliente {

    constructor(nombre,saldo){

        this.nombre = nombre;

        this.saldo = saldo;

    }

    mostrarInformacion(){

        return `Cliente: ${this.nombre} tiene un saldo de $${saldo}`;

    }

}

Lo llamamos:

console.log(juan.mostrarInformacion());

**Propiedades estáticas:**

La diferencia con los métodos comunes es que estas no requieren una instancia para llamarlas ósea no hay que hacer un const juan = new Cliente() . Se pueden llamar directamente con el nombre de la clase:

class Cliente {

    constructor(nombre,saldo){

        this.nombre = nombre;

        this.saldo = saldo;

    }

    mostrarInformacion(){

        return `Cliente: ${this.nombre} tiene un saldo de $${saldo}`;

    }

    static bienvenida(){

        return 'Bienvenido al cajero';

    }

}

console.log(Cliente.bienvenida()); // asi si

const juan = new Cliente('Juan', 400);

console.log(juan.bienvenida()); // asi NO

**Herencia con clases:**

Cuando heredas una clase, se heredan los métodos, las propiedades y el constructor. Acá como hereda el constructor de Cliente no hace falta declararlo en la clase Empresa, y se puede declarar el objeto con los parámetros de Cliente.

class Empresa extends Cliente {

}

const empresa = new Empresa('Codigo con Juan', 400);

Si la nueva clase tiene sus propiedades propias en su constructor se puede usar super() para heredar laspropiedades del padre

class Empresa extends Cliente {

    constructor(nombre, saldo, telefono, categoria){

        super(nombre, saldo);

        this.telefono = telefono;

        this.categoria = categoria;

    }

}

const empresa = new Empresa('Codigo con Juan', 400, 3516341598, 'cursos');

En la clase hija se puede reescribir los métodos.

**Propiedades privadas:**

Public: Se puede acceder dentro de la clase o el objeto

Private: Se puede acceder dentro de la clase

Para hacer las propiedades privadas:

class Cliente {

    #nombre; // esto hace la propiedad privada, se pueden acceder desde los metodos de las clases

    constructor(nombre,saldo){

        this.#nombre = nombre; // Se le pone aca tambien

        this.saldo = saldo;

    }

    mostrarInformacion(){

        return `Cliente: ${this.#nombre} tiene un saldo de $${saldo}`; // cuando lo llamas es con #

    }

    static bienvenida(){

        return 'Bienvenido al cajero';

    }

}

const juan = new Cliente('Juan', 200);

console.log(juan.#nombre); // asi a las privadas no las vas a poder acceder

También se podría hacer con los get y set

class Cliente {

    #nombre; // esto hace la propiedad privada, se pueden acceder desde los metodos de las clases

    setNombre(nombre){

        this.#nombre = nombre;

    }

    getNombre(){

        return this.#nombre;

    }

}

const juan = new Cliente();

juan.setNombre('Juan');

console.log(juan.getNombre);

**Sets, Maps y symbols:**

**Sets:**

Permite crear una lista de valores sin duplicados

Crear un nuevo set:

const carrito = new Set();

Agregar elementos al set:

carrito.add('Camisa');

carrito.add('Disco #1');

carrito.add('Disco #2');

carrito.add('Disco #3');

carrito.add('Disco #3'); // este no se va a agregar, es importante si es mayuscula o minuscula

Saber el length del set

console.log(carrito.size());

Si un elemento existe en el set se busca con .has . devuelve true o false

console.log(carrito.has('Camisa'))

Para borrar elementos lo haces con .delete , también puede devolver true o false

carrito.delete('Disco #3');

Para borrar todos los elementos

carrito.clear();

Son iterables:

carrito.forEach(producto =>{

    console.log(producto);

})

Con esto es muy útil poder eliminar duplicados de un array, si tenemos este array creamos uno nuevo con un new set() entonces se eliminan

const numeros = [10,20,20,30,40,50];

const noDuplicados = new Set(numeros);

console.log(noDuplicados);

**Weakset:**

La diferencia con set es que a este solo se le pueden agregar objetos. En este .size no existe tampoco y tampoco son iterables

const weakset= new WeakSet();

const cliente = {

    nombre: 'Ema',

    saldo: 100

}

weakset.add(cliente);

**Maps:**

Son listas ordenadas en llave y valor, es como un objeto con una sola propiedad. La llave y el valor puede ser cualquier tipo de dato. Tienen mejor performance que un objeto.

const cliente = new Map();

cliente.set('nombre', 'Karen');

cliente.set('tipo', 'premium');

cliente.set('saldo', '3000');

Se puede usar el **size**, y el **has**

Para obtener un valor, en este caso te va a devolver lo que este en nombre, en este caso Karen

console.log(cliente.get('nombre'));

También podes usar el delete

cliente.delete('saldo');

también podes limpiar con cliente.clear();

se puede crear directamente con el constructor:

const paciente = new Map([['nombre', 'paciente'], ['cuarto', 'no definido']])

Para reescribir un valor podes usar el set poniendo la llave ya creada

paciente.set('nombre', 'Raul');

Ahí lo reemplazaría

Se pueden iterar, el index te va a traer la llave ósea el primer valor como ‘nombre’ o ‘cuarto’ y el otro el valor que tiene

paciente.forEach((datos, index)=>{

    console.log(`Llave: ${index}  --  Valor: ${datos}`)

})

**WeakMaps:**

Sirven bien para mantener datos como privados. No se pueden iterar ni conocer su extensión, solo aceptan objetos

const producto = {

    idProducto : 10

}

const weakMap = new WeakMap();

weakMap.set(producto, 'Monitor'); // le asigna a producto monitor

console.log(weakMap.has(producto)); //true

console.log(weakMap.get(producto)); //Monitor no accede a idProducto

console.log(weakMap.delete(producto)); //elimina

**Symbols:**

Ningún symbol es igual es este caso:

const sym = Symbol('1');

const sym2 = Symbol('1');

Las propiedades que usan un symbol no son iterables

const nombre = Symbol();

const apellido = Symbol();

const persona = {};

// Agregar nombre y apellido como llaves del objeto

persona[nombre] = 'Ema'; // en consola aparecen como Symbol()

persona[apellido] = 'Lopez'; // en consola aparecen como Symbol()

persona.tipoCliente = 'Premium';

persona.saldo = 500;

console.log(persona.nombre); //no se va a poder acceder asi

console.log(persona[nombre]); //asi si se va a poder acceder

Definir una descripción del symbol

//Definir una descripcion del symbol

const nombreCliente = Symbol('Nombre del cliente'); // Esta es la descripcion

const cliente = {};

cliente[nombreCliente] = 'Pedro';

**Crear un iterador:**

Como funciona por detrás:  
function crearIterador(carrito){

    let i = 0;

    return {

        siguiente: () =>{

            const fin = (i >= carrito.length);

            const valor = !fin ? carrito[i++] : undefined;

            return {

                fin,

                valor

            }

        }

    }

}

const carrito = ['Producto1', 'Producto2', 'Producto3'];

const recorrerCarrito = crearIterador(carrito);

console.log(recorrerCarrito.siguiente())

console.log(recorrerCarrito.siguiente())

console.log(recorrerCarrito.siguiente())

**Generadores:**

Es una función que retorna un iterador. Yield es una palabra reservada que indica cuales son los valores que se pueden iterar.

function \*crearGenerador(){ // asi se declara un generador

    yield 1;

    yield 'Ema',

    yield 3+3;

    yield true;

}

const iterador = crearGenerador();

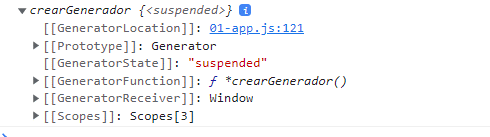
console.log(iterador.next()); // el .next accede al primer valor del generador

Te trae un done y un value a los que podemos acceder



Si hacemos así te va a tirar esto:

console.log(iterador);



Si le das a next una vez después de eso se vuelve a poner en suspended, como que se duerme.

Podes acceder así:

console.log(iterador.next().value)

console.log(iterador.next().done)

Cada vez que escribes next se va a ir al siguiente yield. Entonces:

console.log(iterador.next().value) // va a imprimir 1

console.log(iterador.next().done) // va a imprimir false

console.log(iterador.next().value) // va a imprimir 6 de (3+3) tercer yield

Si lo haces hasta que te quedes sin yields te va a aparecer el generador como closed

Generador con un carrito:

// Generador para carrito de compras

function \*generadorCarrito(carrito){

    for(let i=0; i< carrito.lenght; i++){

        yield carrito[i];

    }

}

const carrito = ['Producto1', 'Producto2', 'Producto3',]

const iterador = generadorCarrito(carrito);

console.log(iterador.next());

console.log(iterador.next());

console.log(iterador.next());

console.log(iterador.next());

**Iteradores nuevos:**

**Entries iterator:**

Agrega una llave si no existe, o imprime llave y valor

const ciudades = ['Londres', 'New York', 'Paris'];

const ordenes = new Set([123,231,131,102]);

const datos = new Map();

datos.set('nombre', 'Ema');

datos.set('profesion', 'Full stack');

for (let entry of ciudades.entries()){

    console.log(entry); // imprime llave y valor

}

for (let entry of ordenes.entries()){

    console.log(entry); // imprime llave y valor que en este caso son el mismo

}

for (let entry of datos.entries()){

    console.log(entry); // imprime llave y valor

}

**Values iterator:**

Te imprimen los valores sin las llaves

//Values

for(let value of ciudades.values()){

    console.log(value); //te imprime los valores del array

}

for(let value of ordenes.values()){

    console.log(value); //te imprime los valores

}

for(let value of datos.values()){

    console.log(value); //te imprime los valores

}

**Keys iterator:**

Te devuelve las llaves:

for(let keys of ciudades.keys()){

    console.log(keys);

}

for(let keys of ordenes.keys()){

    console.log(keys);

}

for(let keys of datos.keys()){

    console.log(keys);

}

**Por default:**

for (let ciudad of ciudades){

    console.log(ciudad);

}

for (let orden of ordenes){

    console.log(orden);

}

for (let dato of datos){

    console.log(dato);

}

**Módulos:**

**IIFE:**

Los módulos se usan para evitar, si estas usando varios archivos js para un html que las variables no se puedan usar desde el otro archivo.

En este caso si lo haces con esta sintaxis a nombreCliente no se podría llamar de otro archivo js vinculado al mismo html.

(function(){

    console.log('Desde un IIFE')

    const nombreCliente = 'Juan';

})();

Si por algún caso queres poder usar las variables en el otro archivo tenes que poner en la variable:

window.nombreCliente = 'Juan';

en vez de CONST

El problema es que se estaría cargando la ventana global

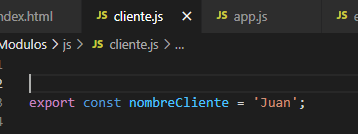
**Módulos:**

**Exportar variables:**

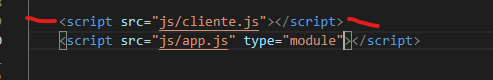
Para no tener que usar el IIFE podemos hacer así

Tenemos dos archivos js cliente y app, desde el cliente queremos enviar una variable para poder ser usada en el app entonces.

En el archivo cliente creamos la variable y le ponemos export



Así ya con esto no vamos a necesitar llamar a este archivo desde el html entonces borramos el script y dejamos el de app



Para que funcione tiene que tener el type=”module”



Después en el app.js tenemos que importar la variable de esta forma:

import { nombreCliente } from './cliente.js';

console.log(nombreCliente);

Si queres exportar mas de uno se pone con coma

import { nombreCliente,ahorro } from './cliente.js';

console.log(ahorro);

console.log(nombreCliente);

**Exportar funciones:**

export function mostrarInformacion(nombre,ahorro){

    return `Cliente ${nombre} - Ahorro: ${ahorro}`;

}

En el otro haces:

import { nombreCliente,ahorro, mostrarInformacion } from './cliente.js';

console.log(ahorro);

console.log(nombreCliente);

console.log(mostrarInformacion(nombreCliente, ahorro));

**Exportar clases:**

export class Cliente{

    constructor(nombre, ahorro){

        this.nombre = nombre;

        this.ahorro = ahorro;

    }

    mostrarInformacion(){

        return `Cliente ${this.nombre} - Ahorro: ${this.ahorro}`;

    }

}

En el otro:

import { nombreCliente,ahorro, mostrarInformacion, tieneSaldo, Cliente } from './cliente.js';

const cliente = new Cliente(nombreCliente, ahorro);

console.log(cliente.mostrarInformacion());

**Heredar una clase exportada:**

Se crea una clase que hereda de Cliente y se exporta:

import {Cliente} from './cliente.js';

export class Empresa extends Cliente{

    constructor(nombre, ahorro, categoria){

        super(nombre, ahorro);

        this.categoria = categoria;

    }

    mostrarInformacion(){

        return `Cliente ${this.nombre} - Ahorro: ${this.ahorro} - Categoría: ${this.categoria}`;

    }

}

Y después se llama en el app.js

import {Empresa} from './empresa.js';

//Importar empresa

const empresa = new Empresa('Codigo con Juan', 100, 'Aprendizaje');

console.log(empresa);

**Export default:**

Solamente puede haber un export default. No hace falta que se ponga dentro de las llaves cuando se importa:

export default function nuevaFuncion(){

    console.log('Este es el export default');

}

Importamos: Se importa con un alias, le podes dar el nombre que vos quieras

import nuevaFuncion, { nombreCliente,ahorro, mostrarInformacion, tieneSaldo, Cliente } from './cliente.js';

import {Empresa} from './empresa.js';

nuevaFuncion();

**IndexedDB:**

Es una base de datos almacenada en el navegador.

**Crear, errores, éxito, configuración:**

document.addEventListener('DOMContentLoaded', () =>{

    crmDB();

})

function crmDB() {

    //Crear base de datos version 1.0

    let crmDB = window.indexedDB.open('crm', 1); //El primer parametro creo que es el nombre de la base

    //Si hay un error

    crmDB.onerror = function(){

        console.log('Hubo un error al crearla'); //puede que el navegador no lo soporte

    }

    // si se creo bien

    crmDB.onsuccess = function() {

        console.log('Base de datos creada');

    }

    // configuracion de la base de datos

    crmDB.onupgradeneeded = function () { // aca creo que se crean las tablas y demas, se ejecuta una vez

        console.log('Este metodo solo se ejecuta una vez');

    }

}

**Creación de las columnas:**

crmDB.onupgradeneeded = function (e) { // aca creo que se crean las tablas y demas, se ejecuta una vez

        const db = e.target.result; //aca obtenes la referencia de la base de datos

        const objectStore = db.createObjectStore('crm', { // Esto nos va a permitir crear las columnas, creo que crea la tabla crm

            keyPath: 'crm',

            autoIncrement: true

        });

        //Definimos las columnas

        objectStore.createIndex('nombre', 'nombre', {unique: false}); // segundo parametro es para consultar

        objectStore.createIndex('email', 'email', {unique: true}); // aca el email debe ser unico

        objectStore.createIndex('telefono', 'telefono', {unique: false});

        console.log('Columnas creadas');

    }

**Insertar registros:**

**El DB tiene**

DB = crmDB.result;

function crearCliente() {

    let transaction = DB.transaction(['crm'] , 'readwrite'); //readonly es otro

    transaction.oncomplete = function () {

        console.log('Transaccion completada');

    }

    transaction.onerror = function() {

        console.log('hubo un error');

    }

    constobjectStore = transaction.objectStore('crm');

    const nuevoCliente = {

        telefono: 12312312,

        nombre: 'Ema',

        email: 'correo@correo.com'

    }

    const peticion = objectStore.add(nuevoCliente); //.put para actualizar y .delete para eliminar

    console.log(peticion);

}

**Leer e imprimir datos de el indexedDB**

El cursor va iterando sobre los elementos

        //Leer el contenido de la base de datos

        const objectStore = DB.transaction('citas').objectStore('citas');

        objectStore.openCursor().onsuccess = function(e) {

            console.log(e.target.result);

        }

Ver proyecto 31

**Ejemplossssss indexedDB::**

**Creación de la base:**

    let DB;

    document.addEventListener('DOMContentLoaded', ()=>{

        crearDB();

    });

    //Crea base de datos de Index DB

    function crearDB(){

        const crearDB = window.indexedDB.open('crm', 1);

        crearDB.onerror = function() {

            console.log('hubo un error');

        }

        crearDB.onsuccess = function () {

            DB = crearDB.result;

        }

        crearDB.onupgradeneeded = function(e) {

            const db = e.target.result;

            const objectStore = db.createObjectStore('crm', {keyPath: 'id', autoincrement:true})

            objectStore.createIndex('nombre', 'nombre', {unique:false});

            objectStore.createIndex('email', 'email', {unique:true});

            objectStore.createIndex('telefono', 'telefono', {unique:false});

            objectStore.createIndex('empresa', 'empresa', {unique:false});

            objectStore.createIndex('id', 'id', {unique:true});

            console.log('DB Lista y creada');

        }

    }

**Agregar elementos a la base de datos y crear alerta:**

    let DB;

    const formulario = document.querySelector('#formulario');

    document.addEventListener('DOMContentLoaded', () =>{

        conectarDB();

        formulario.addEventListener('submit', validarCliente);

    });

    function conectarDB() {

        const abrirConexion = window.indexedDB.open('crm',1);

        abrirConexion.onerror = function() {

            console.log('Hubo un error');

        }

        abrirConexion.onsuccess = function () {

            DB = abrirConexion.result;

        }

    }

    function validarCliente(e) {

        e.preventDefault();

        //Leer todos los inputs

        const nombre = document.querySelector('#nombre').value;

        const email = document.querySelector('#email').value;

        const telefono = document.querySelector('#telefono').value;

        const empresa = document.querySelector('#empresa').value;

        if(nombre === '' || email === '' || telefono === '' || empresa === ''){

            imprimirAlerta('Todos los capos son obligatorios', 'error');

            return;

        }

        //crear un objeto con la informacion

        const cliente = {

            nombre,

            email,

            telefono,

            empresa

        }

        cliente.id = Date.now();

        crearNuevoCliente(cliente);

    }

    function crearNuevoCliente(cliente){

        const transaction = DB.transaction(['crm'], 'readwrite');

        const objectStore = transaction.objectStore('crm');

        objectStore.add(cliente);

        transaction.onerror = function() {

            imprimirAlerta('Hubo un error', 'error');

        }

        transaction.oncomplete = function () {

            imprimirAlerta('El cliente se agregó correctamente');

            setTimeout(()=>{

                window.location.href = 'index.html';

            },3000);

        }

    }

    function imprimirAlerta(mensaje, tipo){

        const alerta = document.querySelector('.alerta');

        if(!alerta){

            //crear la alerta

            const divMensaje = document.createElement('div');

            divMensaje.classList.add('px-4', 'py-3', 'rounded', 'max-w-lg', 'mx-auto', 'mt-6', 'text-center', 'border', 'alerta');

            if(tipo === 'error'){

                divMensaje.classList.add('bg-red-100', 'border-red-400', 'text-red-700');

            }else{

                divMensaje.classList.add('bg-green', 'border-green-400', 'text-green-700');

            }

            divMensaje.textContent = mensaje;

            formulario.appendChild(divMensaje);

            setTimeout(() =>{

                divMensaje.remove();

            },3000);

        }

    }

**Imprimir elementos de la base de datos:**

    document.addEventListener('DOMContentLoaded', ()=>{

        crearDB();

        if(window.indexedDB.open('crm', 1)){

            obtenerClientes();

        }

    });

function obtenerClientes(){

        const abrirConexion = window.indexedDB.open('crm', 1);

        abrirConexion.onerror = function() {

            console.log('Hubo un error');

        }

        abrirConexion.onsuccess = function() {

            DB = abrirConexion.result;

            const objectStore = DB.transaction('crm').objectStore('crm');

            objectStore.openCursor().onsuccess = function(e){

                const cursor = e.target.result;

                if(cursor){

                    const {nombre,empresa,email,telefono, id} = cursor.value;

                    const listadoClientes = document.querySelector('#listado-clientes');

                    listadoClientes.innerHTML += `

                        <tr>

                            <td class="px-6 py-4 whitespace-no-wrap border-b border-gray-200">

                                <p class="text-sm leading-5 font-medium text-gray-700 text-lg  font-bold"> ${nombre} </p>

                                <p class="text-sm leading-10 text-gray-700"> ${email} </p>

                            </td>

                            <td class="px-6 py-4 whitespace-no-wrap border-b border-gray-200 ">

                                <p class="text-gray-700">${telefono}</p>

                            </td>

                            <td class="px-6 py-4 whitespace-no-wrap border-b border-gray-200  leading-5 text-gray-700">

                                <p class="text-gray-600">${empresa}</p>

                            </td>

                            <td class="px-6 py-4 whitespace-no-wrap border-b border-gray-200 text-sm leading-5">

                                <a href="editar-cliente.html?id=${id}" class="text-teal-600 hover:text-teal-900 mr-5">Editar</a>

                                <a href="#" data-cliente="${id}" class="text-red-600 hover:text-red-900">Eliminar</a>

                            </td>

                        </tr>

                    `;

                    cursor.continue();

                }else{

                    console.log('No hay mas registros');

                }

            }

        }

**Edicion:**

(function(){

    let DB;

    let idCliente;

    const nombreInput = document.querySelector('#nombre');

    const emailInput = document.querySelector('#email');

    const telefonoInput = document.querySelector('#telefono');

    const empresaInput = document.querySelector('#empresa');

    const formulario = document.querySelector('#formulario');

    document.addEventListener('DOMContentLoaded', () =>{

        conectarDB();

        //Actualizar el registro

        formulario.addEventListener('submit', actualizarCliente);

        //Verificar el id de la url

        const parametrosURL = new URLSearchParams(window.location.search);

        idCliente = parametrosURL.get('id');

        if(idCliente){

            setTimeout(()=>{

                obtenerCliente(idCliente);

            },100);

        }

    });

    function actualizarCliente(e){

        e.preventDefault();

        if(nombreInput.value === '' || emailInput.value === '' || empresaInput.value === '' || telefonoInput.value ===''){

            imprimirAlerta('Todos los campos son obligatorios', 'error');

            return;

        }

        //Actualizar cliente

        const clienteActualizado = {

            nombre : nombreInput.value,

            email : emailInput.value,

            empresa : empresaInput.value,

            telefono : telefonoInput.value,

            id :  Number(idCliente)

        }

        const transaction = DB.transaction(['crm'], 'readwrite');

        const objectStore = transaction.objectStore('crm');

        objectStore.put(clienteActualizado);

        transaction.oncomplete = function() {

            imprimirAlerta('Editando Correctamente');

            setTimeout(() =>{

                window.location.href = 'index.html';

            },2000);

        }

        transaction.onerror = function() {

            imprimirAlerta('Hubo un error', 'error');

        }

    }

    function obtenerCliente(id){

        const transaction = DB.transaction(['crm'], 'readonly');

        const objectStore = transaction.objectStore('crm');

        const cliente = objectStore.openCursor();

        cliente.onsuccess = function(e) {

            const cursor = e.target.result;

            if(cursor){

                if(cursor.value.id === Number(id)){

                    llenarFormulario(cursor.value);

                }

                cursor.continue();

            }

        }

    }

    function llenarFormulario(datosCliente){

        const {nombre, email, telefono, empresa} = datosCliente;

        nombreInput.value = nombre;

        empresaInput.value = empresa;

        emailInput.value = email;

        telefonoInput.value = telefono;

    }

    function conectarDB(){

        const abrirConexion = window.indexedDB.open('crm',1);

        abrirConexion.onerror = function() {

            console.log('Hubo un error');

        }

        abrirConexion.onsuccess = function () {

            DB = abrirConexion.result;

        }

    }

})();

**Eliminar:**

    function eliminarRegistro(e){

        if(e.target.classList.contains('eliminar')){

            const idEliminar = Number(e.target.dataset.cliente);

            const confirmar = confirm('deseas eliminar este cliente?');

            if(confirmar){

                const transaction = DB.transaction(['crm'], 'readwrite');

                const objectStore = transaction.objectStore('crm');

                objectStore.delete(idEliminar);

                transaction.oncomplete = function(){

                    console.log('Eliminado');

                    e.target.parentElement.parentElement.remove();

                }

                transaction.onerror = function() {

                    console.log('Error')

                }

            }

        }

    }

**Promises, callbacks y programación asíncrona:**

**Callback:**

Si estas descargando un listado de clientes , y mientras estas descargando los últimos 10 clientes se agrega uno nuevo, en ese caso ese ultimo cliente no va a ser parte de mi descarga.

const paises = ['Francia', 'España', 'Portugal', 'Australia', 'Inglaterra'];

function nuevoPais(pais, callback){

    setTimeout(()=>{

        paises.push(pais);

        callback();

    },2000)

}

function mostrarPaises() {

    setTimeout(() =>{

        paises.forEach(pais =>{

            console.log(pais);

        })

    },1000);

}

mostrarPaises();

nuevoPais('Alemania', mostrarPaises);

**CallbackHell:**

Cuando se implementa mal el callback y se hacen muchas funciones por dentro:

const paises = [];

function nuevoPais(pais, callback){

    paises.push(pais);

    console.log(`Agregado: ${pais}`);

    callback();

}

function mostrarPaises(){

    console.log(paises);

}

function iniciarCallbackHell(){

    setTimeout(()=>{

        nuevoPais('Alemania', mostrarPaises);

        setTimeout(() =>{

            nuevoPais('Francia', mostrarPaises);

            setTimeout(()=>{

                nuevoPais('Italia', mostrarPaises);

            },3000)

        },3000)

    },3000);

}

iniciarCallbackHell();

**Promise, .then y .catch:**

El resolve se ejecuta cuando el promise se ejecuta correctamente y el reject cuando tenemos un error en el promise. Se puede aplicar por ejemplo cuando vas a la api a querer descargar algo y si lo trae todo ok ejecuta el resolve y si la api esta caída o algo así el reject.

const aplicarDescuento = new Promise((resolve, reject) =>{

    const descuento = true;

    if(descuento){

        resolve('Descuento aplicado');

    }else{

        reject('No se pudo aplicar el descuento');

    }

})

Si haces un console log a aplicarDescuento podes tener 3 valores:

// hay 3 valores posibles...

// fulfilled - el promise se cumplió

// rejected - el promise no se cumplió

// pending - Aun no se cumple y tampoco fue rechazado

Con el .then decis que vas a llamar a aplicar descuento y **después** vas a hacer algo. Así podremos acceder a la respuesta que se dio en este caso se imprimiría descuento aplicado:

aplicarDescuento

    .then(resultado =>{

        console.log(resultado);

    });

Siempre que el promise falle tenes acceso a un .catch en vez de .then este te va a tirar el mensaje de error de ‘No se pudo aplicar el descuento’ si falló

aplicarDescuento

    .then(resultado =>{

        console.log(resultado);

    })

    .catch(error =>{

        console.log(error);

    })

Se puede poner una función

aplicarDescuento

    .then(resultado =>{

        descuento('descuento aplicado');

    })

    .catch(error =>{

        console.log(error);

    })

function descuento(mensaje){

    console.log(mensaje);

}

Evitar callback hell con promises

const paises = [];

const nuevoPais = pais => new Promise( resolve => {

    setTimeout(() =>{

        paises.push(pais);

        resolve(`Agregado: ${pais}`)

    },3000)

})

nuevoPais('Alemania')

    .then(resultado => {

        console.log(resultado);

        console.log(paises);

        return nuevoPais('Francia'); // llama de nuevo al promise

    })

    .then(resultado =>{

        console.log(resultado);

        console.log(paises);

        return nuevoPais('Inglaterra');

    })

    .then(resultado =>{

        console.log(resultado);

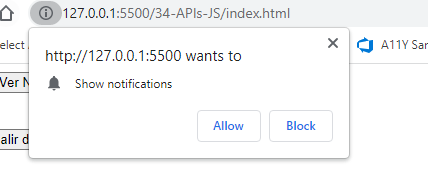
        console.log(pais);

    })

**API’S EN JAVASCRIPT:**

**Notification:**

Cuando te sale el cartelito para permitir las notificaciones



const notificarBtn = document.querySelector('#notificar');

notificarBtn.addEventListener('click', () =>{

    Notification // se usa la api de notificacion

        .requestPermission() //corremos un metodo que pregunta al usuario si quiere dar permisos para recibir notificaciones

        .then(resultado => {

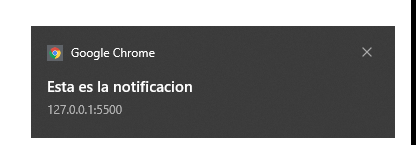
            console.log('El resultado es: ' , resultado);

        })

})

Granted es cuando el usuario acepta las notificaciónes

Acá preguntamos si las acepto y apretando un botón mostramos esta notificación



const verNotificacion = document.querySelector('#verNotificacion');

verNotificacion.addEventListener('click', ()=>{

    if(Notification.permission === 'granted'){

        new Notification('Esta es la notificacion');

    }

})

Modificar notificacion y redirigir a una web:

const verNotificacion = document.querySelector('#verNotificacion');

verNotificacion.addEventListener('click', ()=>{

    if(Notification.permission === 'granted'){

        const notificacion = new Notification('Esta es la notificacion', {

            icon: 'img/ccj.png', //modificamos un par de cosas

            body: 'Codigo con juan'

        });

        notificacion.onclick = function() { //para que te rediriga a una pagina

            window.open('https://codigoconjuan.com');

        }

    }

})

**Intersection Observer:**

Permite identificar cuando un elemento esta visible. Esta observando todo el tiempo. Si vas haciendo scroll y ya ves lo que seleccionaste en la pantalla te marca como isIntersecting true. Tanto si no esta visible o si esta visible. Te va a enviar el cambio a la consola.

document.addEventListener('DOMContentLoaded', () =>{

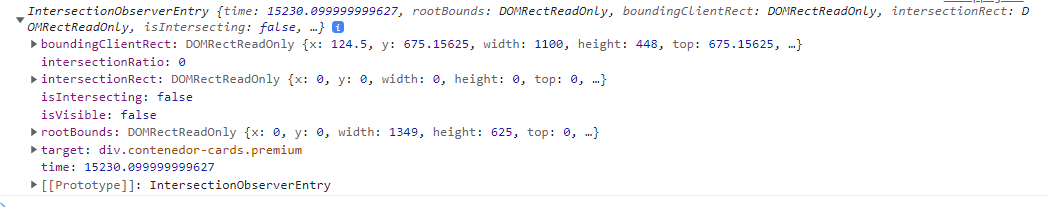
    const observer = new IntersectionObserver(entries => { //lo creamos

        console.log(entries[0]); // te lo muestra en este entry

    });

    observer.observe(document.querySelector('.premium')); //le decimos que elemento va a observar

})



document.addEventListener('DOMContentLoaded', () =>{

    const observer = new IntersectionObserver(entries => { //lo creamos

if(entries[0].isIntersecting){

            console.log('Ya esta visible');

        }

    });

    observer.observe(document.querySelector('.premium')); //le decimos que elemento va a observar

})

**Detectar si hay conexión a internet o no:**

window.addEventListener('online', actualizarEstado);

window.addEventListener('offline', actualizarEstado);

function actualizarEstado(){

    if(navigator.online){ // con este te dice si estas online o no

        console.log('Si estas conectado');

    }else{

        console.log('No estas conectado');

    }

}

**Ejecutar pantalla completa con JS:**

const abrirBtn = document.querySelector('#abrir-pantalla-completa');

const salirBtn = document.querySelector('#salir-pantalla-completa');

abrirBtn.addEventListener('click', pantallaCompleta);

salirBtn.addEventListener('click', cerrarpantallaCompleta);

function pantallaCompleta(){

    document.documentElement.requestFullscreen(); // el documentelement se podría reemplazar por otro elemento

}

function cerrarpantallaCompleta(){

    document.exitFullscreen();

}

**Detectar cuando estamos viendo la pagina web actual:**

document.addEventListener('visibilitychange', () =>{

    console.log(document.visibilityState); // esto te va a traer hidden o visible

    if(document.visibilityState === 'visible'){

        console.log('Ejecutar funcion para ejecutar el video')

    }else{

        console.log('Pausar el video');

    }

})

**Speach recognition API:**

Permite que cuando hables javascript pueda trasladar lo que decis en el micrófono hacia la pagina web.

const salida = document.querySelector('#salida');

const microfono = document.querySelector('#microfono');

microfono.addEventListener('click', ejecutarSpeechAPI);

function ejecutarSpeechAPI() {

    const speechRecognition = webkitSpeechRecognition;

    const recognition = new speechRecognition();

    recognition.start();

    recognition.onstart = function(){ // comienza a grabar

        salida.classList.add('mostrar');

        salida.textContent = 'Escuchando...';

    };

    recognition.onspeechend = function(){ // deja de grabar

        salida.textContent = 'Se dejo de grabar';

        recognition.stop();

    };

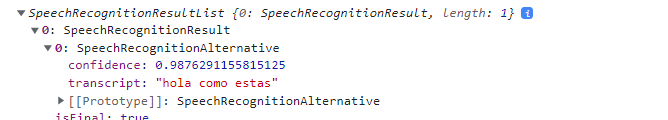
    recognition.onresult = function (e) {

        console.log(e.results);

    }

}

En el transcript esta lo que dijiste por el micrófono



Así te va a traer el confidence y el transcript el confidence es la exactitud de lo que entendió.

    recognition.onresult = function (e) {

        console.log(e.results[0][0]);

    }



**Fetch API:**

Solo puede leer texto plano ,con fetch API podes recibir o enviar datos ya sea de una url, o de un archivo de texto, o json.

Usa promises y con el .then la respuesta te trae el status, para saber si se encontró bien el archivo y esta listo para trabajar

**Si lo haces con archivo de text:**

const cargarTxtBtn = document.querySelector('#cargarTxt');

cargarTxtBtn.addEventListener('click', obtenerDatos);

function obtenerDatos() {

    const url = 'data/datos.txt'; // de donde vienen los datos

    fetch(url)

        .then(respuesta => {

            console.log(console.log(respuesta));

        })

}

Propiedades:

    fetch(url)

        .then(respuesta => {

            console.lo(respuesta);

            console.lo(respuesta.status); // 200 404 etc

            console.lo(respuesta.statusText); // este devuelve OK si esta bien

            console.lo(respuesta.url); //url del archivo

            console.lo(respuesta.type); // basic

        })

En el return podes usar text o json dependiendo del archivo que este llamando, en este caso para traer el contenido se usa text() en el return:

    fetch(url)

        .then(respuesta => {

            console.lo(respuesta);

            console.lo(respuesta.status); // 200 404 etc

            console.lo(respuesta.statusText); // este devuelve OK si esta bien

            console.lo(respuesta.url); //url del archivo

            console.lo(respuesta.type); // basic

            return respuesta.text();

        });

Con el segundo then ya podes obtener el contenido del archivo y en el catch va a llegar si hay un error en el nombre de la url o algo mas:

    fetch(url)

        .then(respuesta => {

            console.lo(respuesta);

            console.lo(respuesta.status); // 200 404 etc

            console.lo(respuesta.statusText); // este devuelve OK si esta bien

            console.lo(respuesta.url); //url del archivo

            console.lo(respuesta.type); // basic

            return respuesta.text();

        })

        .then(datos => { // en este segundo then ya viene el contenido del archivo

            console.log(datos);

        })

        .catch(error =>{ // por si hay un errorcae aca

            console.log(error);

        })

**Si lo haces con un .json (Objeto):**

Cuando lo imprimis en consola te convierte el json en un objeto. En este ejemplo también lo reemplazamos en el html con innerHTML.

const cargarJSONBtn = document.querySelector('#cargarJSON');

cargarJSONBtn.addEventListener('click', obtenerDatos);

function obtenerDatos() {

    const url = 'data/empleado.json';

    fetch(url)

        .then(respuesta =>{

            console.log(respuesta)

            return respuesta.json();

        })

        .then(resultado =>{

            mostrarHTML(resultado);

        })

        .catch(error =>{

            console.log(error);

        })

}

function mostrarHTML({empresa, id, nombre, trabajo}){

    const contenido = document.querySelector('.contenido');

    contenido.innerHTML = `

        <p>Empleado ${nombre}</p>

        <p>Id ${id}</p>

        <p>Empresa ${empresa}</p>

        <p>trabajo ${trabajo}</p>

    `;

}

**Con array de objetos JSON:**

const cargarJSONArrayBtn = document.querySelector('#cargarJSONArray');

cargarJSONArrayBtn.addEventListener('click', obtenerDatos);

function obtenerDatos() {

    const url = 'data/empleados.json';

    fetch(url)

        .then(respuesta => respuesta.json())

        .then(resultado => mostrarHTML(resultado))

}

function mostrarHTML(empleados){

    const contenido = document.querySelector('.contenido');

    let html = '';

    empleados.forEach(empleado => {

        const {id, nombre, empresa, trabajo} = empleado;

        html += `

            <p>Empleado ${nombre}</p>

            <p>Id ${id}</p>

            <p>Empresa ${empresa}</p>

            <p>trabajo ${trabajo}</p>

        `

    });

    contenido.innerHTML = html;

}

**Spinner de carga cuando hace una llamada a la API:**

function Spinner() {

    limpiarHTML();

    const divSpinner = document.createElement('div');

    divSpinner.classList.add('sk-fading-circle');

    divSpinner.innerHTML = `

    <div class="sk-circle1 sk-circle"></div>

    <div class="sk-circle2 sk-circle"></div>

    <div class="sk-circle3 sk-circle"></div>

    <div class="sk-circle4 sk-circle"></div>

    <div class="sk-circle5 sk-circle"></div>

    <div class="sk-circle6 sk-circle"></div>

    <div class="sk-circle7 sk-circle"></div>

    <div class="sk-circle8 sk-circle"></div>

    <div class="sk-circle9 sk-circle"></div>

    <div class="sk-circle10 sk-circle"></div>

    <div class="sk-circle11 sk-circle"></div>

    <div class="sk-circle12 sk-circle"></div>

    `;

    resultado.appendChild(divSpinner);

}

Lo llamamos antes del Fetch

function consultarAPI(ciudad, pais){

    const appId = '653bf7c58be72a546bf4862c3c903e5d';

    const url = `https://api.openweathermap.org/data/2.5/weather?q=${ciudad},${pais}&appid=${appId}`;

    Spinner(); //Muestra el spinner

    fetch(url)

        .then(respuesta => respuesta.json())

        .then(datos => {

            limpiarHTML();

            if(datos.cod === '404'){

                mostrarError('Ciudad no encontrada');

                return;

            }

            //Imprime la respuesta en HTML

            mostrarClima(datos);

        })

}

**Async Await:**

**Try catch:**

Se usa mucho try catch con async await. En javascript cuando el código falla deja de ejecutarse, como en este ejemplo. Esa función hola falla entonces no se ejecuta el console log de abajo

console.log(1+1);

hola();

console.log(2+2);

Eso lo soluciona el try catch, que intenta ejecutar una pieza de código y si no se puede se obtiene una excepción como un mensaje de error. Como si consumis una api y si no se puede se le puede enviar un mensaje al usuario, y el resto del código sigue funcionando correctamente

Así se solucionaría:

console.log(1+1);

try{

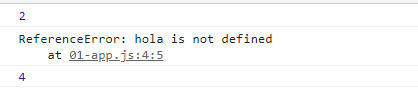
    hola();

} catch (error){

    console.log(error);

}

console.log(2+2);



El try catch se suele usar para conectar a una base de datos, consultar info de una API , autenticar un usuario o acciones que en caso de que falle la app siga funcionando.

Async es la función en este caso ejecutar() . En este caso await llama a descargarClientes, pero dice voy a esperar que finalice la función entera de descargarClientes() ósea espera esos 3 segundos del setTimeOut. Await **retiene la ejecución del código hasta que se resuelve el promise**

El console.log 2+2 no se ejecuta hasta que termina de ejecutarse la función de descargarClientes()

**Con function declaration:**

function descargarClientes(){

    return new Promise((resolve,reject) =>{

        const error = false;

        setTimeout(() =>{

            if(!error){

                resolve('El listado de clientes se descargo correctamente');

            }else{

                reject('Error en la conexion');

            }

        },3000)

    })

}

//Async await

async function ejecutar() {

    try{

        console.log(1+1);

        const respuesta = await descargarClientes();

        console.log(2+2);

        console.log(respuesta); // aca te va a mostrar el mensaje de error o exito

    }catch(error){

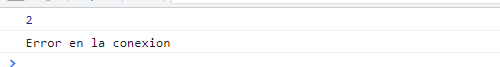
        console.log(error);

    }

}

ejecutar();

**Si en el await detecta un reject pasa automáticamente al catch.** En ese caso si el error fuera true, entra al reject se ejecuta el console.log(1+1) pero después sale del try y entra al catch, en el (error) creo que le pasa el string de ‘Error en la conexión’ de la función de descargarCliente



**Con function expresión:**

Así sería la sintaxis

const ejecutar = async() =>{

    try{

        console.log(1+1);

        const respuesta = await descargarClientes();

        console.log(2+2);

        console.log(respuesta); // aca te va a mostrar el mensaje de error o exito

    }catch(error){

        console.log(error);

    }

}

ejecutar();

**Manejar múltiples awaits:**

Se podría hacer de esta manera pero tiene un error:

function descargarNuevosClientes() {

    return new Promise((resolve)=>{

        console.log('Descargando clientes...');

        setTimeout(()=>{

            resolve('Los clientes fueron descargados');

        },5000)

    })

}

function descargarNuevosPedidos() {

    return new Promise((resolve)=>{

        console.log('Descargando pedidos...');

        setTimeout(()=>{

            resolve('Los pedidos fueron descargados');

        },3000)

    })

}

const app = async ()=>{

    try {

        const clientes = await descargarNuevosClientes();

        console.log(clientes);

        const pedidos = await descargarNuevosPedidos();

        console.log(pedidos);

    } catch (error) {

        console.log(error);

    }

}

app();

El error es que como descargarNuevosClientes tarda 5 segundos y descargarNuevosPedidos tarda 3 segundos, cuando se hace el primer await va a frenar la descarga de pedidos por 5 segundos, y después se va a ejecutar el de pedidos.

Mejor: se puede hacer con Promise.all() así se van a ejecutar las dos funciones al mismo tiempo. Y vas a ganar mucho en performance

const app = async ()=>{

    try {

        const respuesta = await Promise.all([descargarNuevosClientes(), descargarNuevosPedidos()]);

        console.log(respuesta[0]);

        console.log(respuesta[1]);

    } catch (error) {

        console.log(error);

    }

}

app();

**await async con api:**

Aca se muestra como se hace con fetch y con async await, con await async es un poco mas larga pero es mejor

const url = 'https://picsum.photos/list';

document.addEventListener('DOMContentLoaded', obtenerDatos);

//para hacerlo con promises

function obtenerDatos(){

    fetch(url)

        .then(respuesta => respuesta.json())

        .then(resultado => console.log(resultado))

        .catch(error => console.log(error));

}

//para hacerlo con await async

async function obtenerDatos(){

    try {

        const respuesta = await fetch(url);

        const resultado = await respuesta.json();

        console.log(resultado)

    } catch (error) {

        console.log(error);

    }

}

**Cambiar proyecto de pixabay de fetch a async await:**

async function buscarImagenes() {

    const termino = document.querySelector('#termino').value;

    const key = '22785665-c713d451bf1a8aeedc1104e8b';

    const url = `https://pixabay.com/api/?key=${key}&q=${termino}&per\_page=${registrosPorPagina}&page=${paginaActual}`;

    // fetch(url)

    //     .then(respuesta => respuesta.json())

    //     .then(resultado => {

    //         totalPaginas = calcularPaginas(resultado.totalHits);

    //         mostrarImagenes(resultado.hits);

    //     })

    try {

        const respuesta = await fetch(url);

        const resultado = await respuesta.json();

        totalPaginas = calcularPaginas(resultado.totalHits);

        mostrarImagenes(resultado.hits);

    } catch (error) {

        console.log(error);

    }

}

**JSON-SERVER:**

**Instalar json-server:**

Json-server permite simular la creación de una api

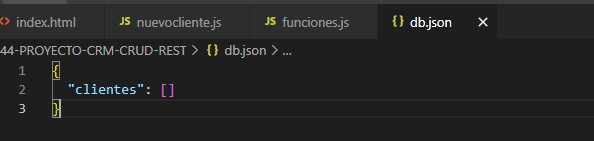
En el powershell ejecutar como admin, correr npm install -g json-server

Para correrlo es con:

npx json-server --watch db.json no lo probe pero después pone --p 4000 para asignar el puerto

**Rutas:**

Si en el db.json tenes estos clientes: las rutas van a ser tanto para get,post, etc con /cliente



****

**CRUD APIS:**

**Agregar datos a la API:**

En este caso cuando queremos agregar un nuevo cliente hacemos así: se hace el fetch a la url y se crea un objeto con el tipo de llamada que va a ser un POST, y el body, que el body se envia de dos formas como string o como objecto, en este caso lo hacemos como string poniendo JSON.stringify(objeto); y después le pones los headers que tiene info de que tipo de datos estamos enviando, este varía según el tipo de dato que envíes al servidor.

const url = 'http://localhost:3000/clientes';

export const nuevoCliente = async(cliente)=>{

    try {

        await fetch(url, {

            method: 'POST',

            body: JSON.stringify(cliente),

            headers: {

                'Content-Type': 'application/json'

            }

        });

        window.location.href = 'index.html'; //esto se ejecuta si se trae bien el cliente

    } catch (error) {

        console.log(error);

    }

}

**Obtener datos:**

Cuando queres traer datos de la API Por defecto fetch envía un get así que no habría que especificarlo:

const resultado = await fetch(url);

Quedaría asi: Usando la misma URL que se uso para el post

export const obtenerClientes = async() => {

    try {

        const resultado = await fetch(url);

        const clientes = await resultado.json();

        return clientes;

    } catch (error) {

        console.log(error);

    }

}

**Eliminar datos:**

//Elimina un cliente

export const eliminarCliente = async id => {

    try {

        await fetch(`${url}/${id}`,{

            method: 'DELETE'

        })

    } catch (error) {

        console.log(error);

    }

}

Le pone un evento click al botón y llama a esta función:

    function confirmarEliminar(e) {

        if(e.target.classList.contains('eliminar')){

            const clienteId = parseInt(e.target.dataset.cliente);

            const confirmar = confirm('Deseas eliminar el registro?');

            if(confirmar){

                eliminarCliente(clienteId);

            }

        }

    }

**Editar:**

Buscamos los datos de la API y los pegamos en los inputs para que la persona lo pueda editar:

    //Campos del formulario

    const nombreInput = document.querySelector('#nombre');

    const emailInput = document.querySelector('#email');

    const empresaInput = document.querySelector('#empresa');

    const telefonoInput = document.querySelector('#telefono');

    const idInput = document.querySelector('#id');

    document.addEventListener('DOMContentLoaded', async () => {

        const parametrosURL = new URLSearchParams(window.location.search);

        const idCliente = parseInt(parametrosURL.get('id')); //Saca el parametro id de la url

        const cliente = await obtenerCliente(idCliente); //busca el cliente x id

        mostrarCliente(cliente);

    });

    function mostrarCliente(cliente){

        const {nombre,empresa,email,telefono,id} = cliente;

        nombreInput.value = nombre;

        empresaInput.value = empresa;

        emailInput.value = email;

        telefonoInput.value = telefono;

        idInput.value = id;

    }

export const obtenerCliente = async id =>{

    try {

        const resultado = await fetch(`${url}/${id}`);

        const cliente = await resultado.json();

        return cliente;

    } catch (error) {

        console.log(error);

    }

}

El export es xq usaba módulos en el ejemplo.

**Functional JS:**

Es crear tu código usando funciones, es diferente a lo que veníamos haciendo, hay reglas como que las funciones tienen que tomar una entrada y tener una salida de datos, no se permite la modificación de los datos. Tiene una sintaxis mas matemática

Conceptos calve:

* Inmutabilidad: los datos no deben modificarse (usar const casi siempre)
* Separar funciones de datos: Se usan mucho funciones que retornan un nuevo dato a array method, así tenes funciones que entregan un resultado nuevo pero nunca modifican los datos
* First-class functions: Es poder crear funciones que parezcan cualquier variable como lo es function expresssion es algo así:

Const suma = function(a,b) {

Return a + b;

}

Const resultado = suma;

**First class function:**

const suma = function (a,b) { //esto es una funcion

    return a+b;

}

const resultado = suma; // aca en vez de asignarle 4 le asignas la funcion

console.log(resultado(10,20)) // haces asi para llamarla

**Pasar funciones como argumentos:**

const suma = (a,b) => a+b; //creas la funcion

const multiplicar = (a,b) => a\*b; //creas la funcion

const sumarOmultiplicar = fn => fn(10, 20); // aca le pasas los valores

console.log(sumarOmultiplicar( suma )) // pasas la funcion como parametro

console.log(sumarOmultiplicar( multiplicar )) // pasas la funcion como parámetro

**Separar los datos de las funciones, higher order function:**

const carrito = [

    { nombre: 'Monitor 20 Pulgadas', precio: 500},

    { nombre: 'Televisión 50 Pulgadas', precio: 700},

    { nombre: 'Tablet', precio: 300},

    { nombre: 'Audifonos', precio: 200},

    { nombre: 'Teclado', precio: 50},

    { nombre: 'Celular', precio: 500},

    { nombre: 'Bocinas', precio: 300},

    { nombre: 'Laptop', precio: 800},

];

//Para filtrar los que sean mayor a 400 sería asi:

const resultado = carrito.filter(producto => {

    return producto.precio > 400

})

console.log(resultado);

//Con higher order function sería  asi

const mayor400 = producto =>{ // esto es una funcion

    return producto.precio > 400;

}

const resultado2 = carrito.filter(mayor400); // aca le pasamos la funcion al filter como parametro

console.log(resultado2);

**Usando map con functional JS:**

Recordar que con map, creamos un nuevo array sin modificar el original.

const carrito = [

    { nombre: 'Monitor 20 Pulgadas', precio: 500},

    { nombre: 'Televisión 50 Pulgadas', precio: 700},

    { nombre: 'Tablet', precio: 300},

    { nombre: 'Audifonos', precio: 200},

    { nombre: 'Teclado', precio: 50},

    { nombre: 'Celular', precio: 500},

    { nombre: 'Bocinas', precio: 300},

    { nombre: 'Laptop', precio: 800},

];

const obtenerNombres = producto => {

   return producto.nombre; // con esta fn vamos devolviendo cada uno de los nombres del array

}

const resultado = carrito.map(obtenerNombres); // aca con el map llamamos a la funcion

console.log(resultado);

**Menor cantidad de código en las funciones:**

const obtenerNombres = p => p.nombre;

const resultado = carrito.map(obtenerNombres);

console.log(resultado);

///////////////

const mayor400 = p =>p.precio > 400;

const resultado2 = carrito.filter(mayor400);

console.log(resultado2);

**Pure functions:**

Son funciones que retornan un dato, pero no modifican los valores de las variables. Retornan un dato nuevo, y con un parámetro deben retornan la misma cantidad de datos que recibe la entrada.

const duplicar = (numero) => { //toma UN numero y tiene que retornar UN numero

    return numero \* 2;

}

const numero1 = 20; //numero1 mantiene su valor

const resultado = duplicar(numero1); // el resultado se va a guardar en resultado

console.log(resultado); //40

console.log(numero1);  //20

**Funciones que retornan funciones:**

const obtenerCliente = () => () => console.log('Ema'); // es una funcion que retorna otra funcion

const fn = obtenerCliente(); // fn entonces sería () => console.log('Ema');

fn(); // aca va a ejecutar el console.log('Ema');

**Closures:**

Si creas una constante dentro de una función el scope de esa constante solo va a ser dentro de la función, para poder usarla fuera de la función se usa closure.

const obtenerCliente = () => {

    const nombre = 'Juan'; // la variable existe aca

    function muestraNombre() {

        console.log(nombre);

    }

    return muestraNombre; //retorna la funcion con el console.log

}

const cliente = obtenerCliente(); //guardamos la funcion en una variable

cliente(); //ejecutamos la funcion , ACA SI SE VA A IMPRIMIR Juan

**Partials y curryng:**

Curryng es dividir una función que toma mas de un parámetro en argumentos de forma parcial

const suma = (a,b,c) => a + b + c;

const parcial = a => (b,c) => suma(a,b,c); //funcion dentro de funcion

const primerNumero = parcial(5); // aca le pasamos la primera parte que sería (a)

const resultado = primerNumero(4,3) // a lo que sería la funcion primerNumero creada se les pasa los otros dos parametros a y b

console.log(resultado); // 12

También se puede hacer asi para dividirlo en 3 en vez de dos:

////////Se podría hacer asi tambien, lo dividimos en 3 partes:

parcial = a => b => c => suma(a,b,c); //no lleva el const xq ya estaba creado

primerNumero = parcial(5) //no lleva el const xq ya estaba creado

const segundoNumero = primerNumero(4);

resultado = segundoNumero(8);

//en vez de hacer esto de aca arriba se debería hacer:

const resultadoParcial = parcial(5)(4)(3); //tambien va a dar 12

console.log(resuresultadoParcialltado);

**Composition:**

Es como una alternativa a las clases, se usa cuando tenes funciones que se pueden compartir entre objetos.

const obtenerNombre = info => ({

    mostrarNombre(){

        console.log(`Nombre: ${info.nombre}`);

    }

})

const guardarEmail = info => ({

    agregarEmail(email){

        console.log(`Guardando email en: ${info.nombre}`);

        info.email = email;

    }

})

const obtenerEmail = info => ({

    mostrarEmail() {

        console.log(`Correo: ${info.email}`);

    }

})

const obtenerEmpresa = info => ({

    mostrarEmpresa(){

        console.log(`Empresa: ${info.empresa}`)

    }

})

const obtenerPuesto= info => ({

    mostrarPuesto(){

        console.log(`Puesto: ${info.puesto}`)

    }

})

function Cliente(nombre, email, empresa) {

    let info = {

        nombre,

        email,

        empresa

    }

    return Object.assign(

        info,

        obtenerNombre(info),

        guardarEmail(info),

        obtenerEmail(info),

        obtenerEmpresa(info)

    )

}

function Empleado(nombre, email, puesto) {

    let info = {

        nombre,

        email,

        puesto

    }

    return Object.assign(

        info,

        obtenerNombre(info),

        guardarEmail(info),

        obtenerEmail(info),

        obtenerPuesto(info)

    )

}

const cliente = Cliente('Juan', null, 'lalala')

cliente.mostrarNombre();

cliente.agregarEmail('cliente@asadd.com');

cliente.mostrarEmail();

cliente.mostrarEmpresa();

const empleado = Empleado('Pedro', null, 'programador')

empleado.mostrarNombre();

empleado.agregarEmail('empleado@asdas.com');

empleado.mostrarEmail();

empleado.mostrarPuesto();

**Scope:**

Si se crea una variable dentro de una función el scope es solo dentro de esa función. Si haces una global se podrá usar dentro de las funciones, pero si creas dos va a tener prioridad la que este dentro de la función:

Acá imprimiría Pedro:

const cliente = 'Ema';

function mostrarCliente() {

    const cliente = 'Pedro';

    console.log(ciente);

}

mostrarCliente();

**Hoisting:**

Existen 2 fases en javascript una de creación y otra de ejecución.

En la primera se crean todas las variables y funciones, se registran y en la segunda se ejecutan.

Cual es la diferencia entre function expression y function declaration?

**Function declaration:**

obtenerCleinte('Ema'); // esta se puede llamar tanto antes como despues de la funcion creada

function obtenerCleinte(nombre) { //Function declaration

    console.log(`El nombre del cliente es: ${nombre}`)

}

obtenerCleinte('Ema'); // esta se puede llamar tanto antes como despues de la funcion creada

**Function expression:**

En este caso marca error porque primero se define como undefined obtenerCliente2, y después en la segunda vuelta recién le asigna el valor de la función.

obtenerCliente2('Lopez'); //aca NO funciona

const obtenerCliente2 = function(nombre){

    console.log(`El nombre del cliente es: ${nombre}`)

}

obtenerCliente2('Lopez'); //aca funciona

**Coercion:**

Es la conversion automática o implícita de valores de un tipo dado a otro

**Cohercion implícita:**

const numero1 = 20;

const numero2 = "40";

console.log(numero1 + numero2); //imprime 2040 lo convierte a string esto es COHERCION IMPLICITA

**Cohercion explicita:**

Cuando convertís los valores, cuando usas Number, o toString(), JSON.stringify() etc.

console.log(Number(numero2)); // aca se usa una funcion para modificarlo es COHERCION EXPLICITA

**Implicit Binding:**

Se indica de una forma clara con la palabra this donde encontrar sus valores. En este caso tenemos que decirle en que lugar va a encontrar las variables de nombre y edad. Entonces usamos el this. Para decirle que va a encontrar las variables en ese mismo objeto. En el caso de la mascota el this se va a referir al mismo objeto de mascota y no se va a mezclar con los de la clase superior.

const usuario = {

    nombre: 'Juan',

    edad: 20,

    informacion() {

        console.log(`Mi nombre es ${this.nombre} y mi edad es ${this.edad}`);

    },

    mascota: {

        nombre: 'Hook',

        edad:1,

        informacion() {

            console.log(`Mi nombre es ${this.nombre} y mi edad es ${this.edad}`);

        }

    }

}

usuario.informacion();

usuario.mascota.informacion();

**Explicit Binding:**

Se usan 3 funciones, una llamada call, otra apply y una llamada bind.

**Call** existe en todas las funciones de javascript incluso en las que uno crea, le podes pasar un objeto o hasta una función. Cuando se usa.call y pasas un arreglo se tiene que pasar cada elemento de forma individual con su posición en el arreglo.

En este caso creamos una función, pero dentro llamamos a variables que están fuera dentro de un objeto y dentro de un array, entonces con el nombre de la función y call “agregamos” los parámetros

function persona(el1, el2){

    console.log(`Mi nombre es ${this.nombre} y escucho ${el1} y ${el2}`); // el this.nombre es implicit los otros no

}

const informacion = {

    nombre: 'Juan'

}

const musicaFavorita = ['Heavy metal', 'Rock'];

persona.call(informacion, musicaFavorita[0], musicaFavorita[1]);

**Apply** Con este si estas pasando un arreglo no tenes que pasar cada elemento individual, sino que podes pasar directamente el arreglo:

persona.apply(informacion, musicaFavorita);

**Bind** Acá si se tienen que pasar los elementos de forma individual pero pero la diferencia es que te crea una nueva función.

const nuevaFuncion = persona.bind(informacion, musicaFavorita[0], musicaFavorita[1]);

nuevaFuncion();

**New Binding:**

Creas un nuevo objeto con el object constructor, es como C#.

function Auto(modelo,color){

    this.modelo = modelo; // este es el new Binding

    this.color = color; // este es el new Binding

}

const auto = new Auto('Camaro', 'Negro');

**Windows Binding:**

window.color = 'Negro'; // asi asignas una variable a la ventana global

function hola() {

    console.log(color);

}

hola();

**Event Loop:**

El código de javascript es de un solo hilo, ósea que solamente sucede una cosa a la vez, cuando se completa una cosa se llama a la siguiente. Eso es el loop de eventos, mientras haya tareas va a estar ejecutándose.

En javascript hay eventos que tienen mas prioridad que otros,

console.log('Primero'); // 1

setTimeout(() => {

    console.log('Segundo'); // 5

}, 0);

console.log('Tercero'); // 2

setTimeout(() => {

    console.log('Cuarto'); // 6

}, 0)

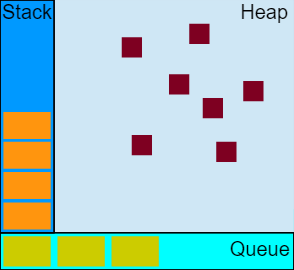
new Promise(function(resolve){

    resolve('Desconocido...') // 4

}).then(console.log)

console.log('Ultimo'); // 3

Acá el comentario es el orden en el que se imprimen las cosas.



Las funciones y el código se va colocando en el stack, dependiendo de la naturaleza de cada función, y otras se colocan en el Queue. Los console.log en el caso anterior se ve que se colocan directamente en el Stack, el setTimeOut y el promise se terminan yendo al Queue. Una vez que se terminan de ejecutar los console.log javascript saca del Queue los promises y los pasa al Stack , no pasa los setTimeOut aun xq parece que el promise tiene mas prioridad, entonces ejecuta los promises, después hace lo mismo con los setTimeOut que quedan.

Si al código le agregamos una función se va a ejecutar antes que el promise.

**Self:**

Se refiere a la ventana global y es lo mismo que window. También lo usan para reemplazar el this pero es lo mismo.

self.onload = () => { //es lo mismo que window

    console.log('Ventana Lista')

}

window.nombre = 'Monitor 20 pulgadas';

const producto = {

    precio: 30,

    disponible: true,

    mostrarInfo: function() {

        const self = this;

        return `El producto: ${self.nombre} tiene un precio de ${self.precio}` // aca tambien es lo mismo cambia el nombre nomas

    }

}

console.log(producto.mostrarInfo());

**Progressive Web Application (PWA) y Service Workers: NO ME FUNCIONO**

**Caracteristicas de una PWA:**

* Rapida – Cargan toda la info en menos de 5 seg.
* Instalable – Se puede navegar o instalar en tu navegador o teléfono móvil como una aplicación nativa
* Soporte offline – Pueden funcionar incluso sin conexión a internet.

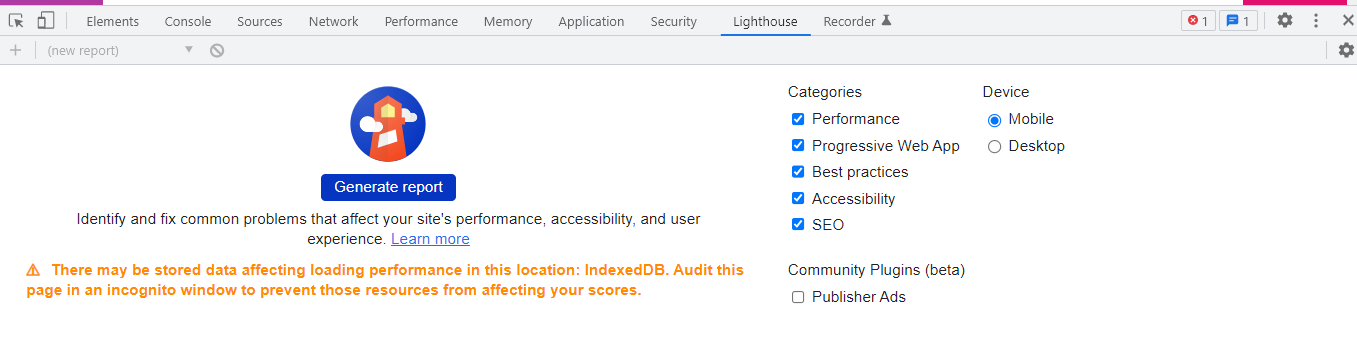
**Service Workers:**

* Son la base de una PWA. Son scripts que están corriendo todo el tiempo detrás de escenas.
* Funcionan offline
* No tienen acceso al DOM
* Cargan de forma instantánea
* Pueden sincronizar datos detrás de escena o sin interferir en la navegación

**Funciones no disponibles en Service Workers:**

* Window (usa self)
* Document (usa caches)
* Local Storage (se utiliza fetch)

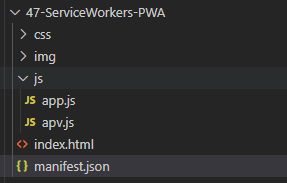
Primero va a esta pestaña



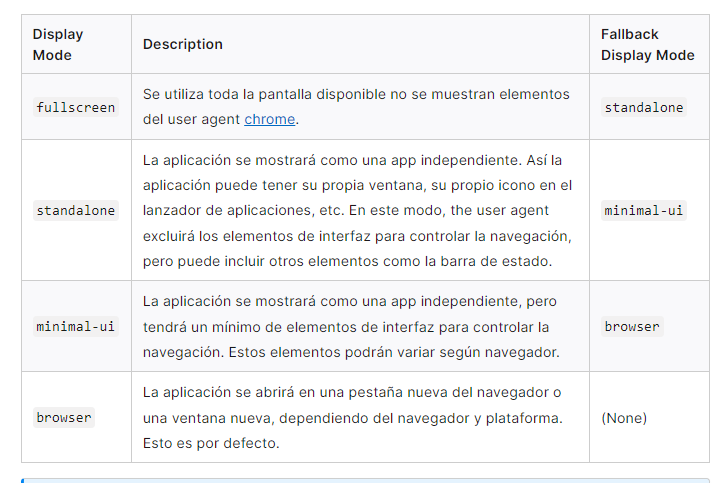
Con eso vas a poder sacar una estadística de los puntos a la derecha,

**PWA:**

Para empezar con PWA creamos un archivo llamado manifest.json



Se pone dentro del manifest el nombre del proyecto que va a ser APV en este caso, la start url va a ser en donde inicia, en el display va a ser como se va a ver la interfaz. Los valores de display pueden ser:



Acá ponemos standalone para que no tenga la barra de navegación una vez que lo instalemos, el orientation portrait es que va a ser para móvil y las imágenes son los iconos de la carpeta png

{

    "name": "APV",

    "short\_name": "APV",

    "start\_url": "/index.html",

    "display": "standalone",

    "background\_color": "#D41872",

    "theme\_color": "#D41872",

    "orientation": "portrait",

    "icons": [

        {

          "src": "img/icons/Icon-72",

          "type": "image/png",

          "sizes": "72x72"

        },

        {

          "src": "img/icons/Icon-120.png",

          "type": "image/png",

          "sizes": "120x120"

        },

        {

          "src": "img/icons/Icon-128.png",

          "type": "image/png",

          "sizes": "128x128"

        },

        {

          "src": "img/icons/Icon-144.png",

          "type": "image/png",

          "sizes": "144x144"

        },

        {

          "src": "img/icons/Icon-152.png",

          "type": "image/png",

          "sizes": "152x152"

        },

        {

          "src": "img/icons/Icon-196.png",

          "type": "image/png",

          "sizes": "196x196"

        },

        {

          "src": "img/icons/Icon-256.png",

          "type": "image/png",

          "sizes": "256x256"

        },

        {

          "src": "img/icons/Icon-512.png",

          "type": "image/png",

          "sizes": "512x512"

        }

      ]

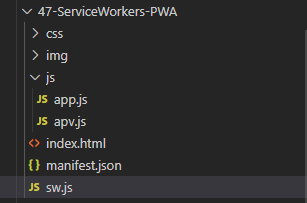
}

En el html hay que poner para que lea la info

    <link rel="manifest" href="manifest.json">

**Service Worker:**

En el raíz del proyecto creamos un js llamado sw.js



En el app.js preguntamos si el navegador lo soporta a serviceWorker y si lo soporta lo registramos

if('serviceWorker' in navigator){

    navigator.serviceWorker.register('./sw.js')

    .then(registrado => console.log('Se instalo correctamente', registrado))

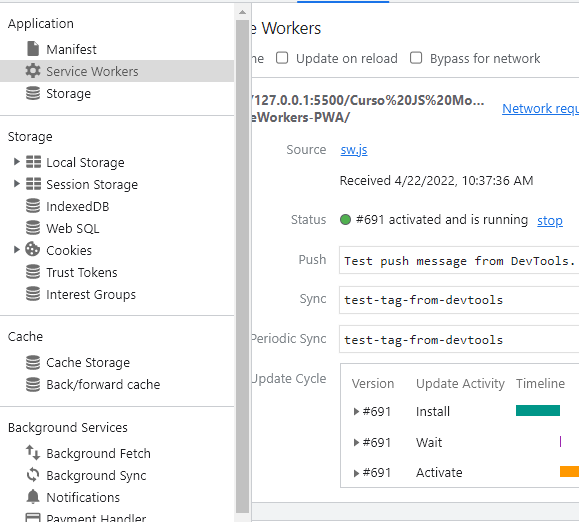
    .catch( error => console.log('Fallo la instalacion', error));

} else{

    console.log('No lo soporta');

}

Podemos controlar que esta registrado acá:



Instalar y activar service worker:

En el sw.js

//Cuando se instala el service Worker, solo se ejecuta una vez

self.addEventListener('install', e => {

    console.log('Instalado el Service Worker');

    console.log(e);

})

//activar serviceWorker

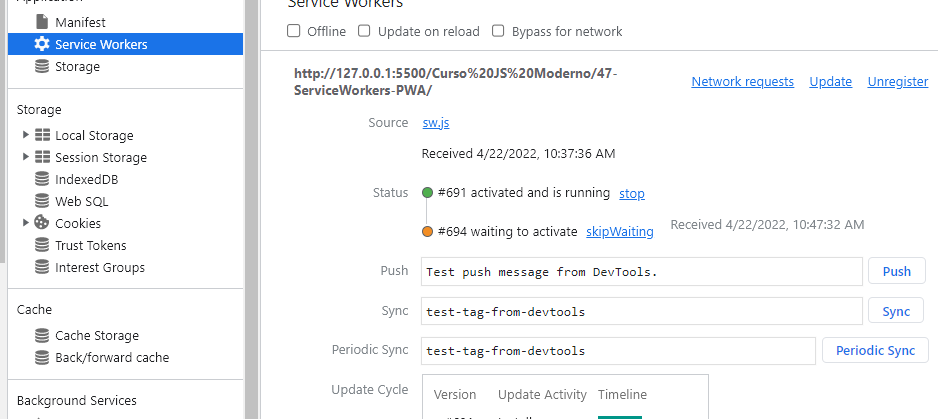
self.addEventListener('activate', e => {

    console.log('Service Worker Activado');

    console.log(e);

})

Para activarlo hay que ir a inspeccionar y darle click en skip waiting



Hacer una PWA instalable

En el sw.js

//Evento fetch para descargar archivos estaticos

self.addEventListener('fetch', e => {

    console.log('Fetch...', e);

})

La pestaña debe estar sola la de la app en Chrome, me debería aparecer un cartelito para instalar. Si no aparece ir a los 3 puntitos, mar herramientas y crear shortcut y crear

Se va a ver como si fuera una aplicación de Chrome



Esto lo podes usar como una aplicación para el celu si tenes un emulador lo podes ver. Pero en el emulador la ip es distinta, es 10.0.2.2:5500/index.html

Entonces:

En los 3 puntitos debería aparecer un remote devices pero no ta

Cachear archivos:

No es bueno cachear todas las paginas xq sino se haría muy lento. Se pueden cachear los archivos para que la app funcione sin conexión a internet.

Al sw.js le agregamos:

const nombreCache = 'apc-v1';

const archivos = [

    '/',

    'index.html',

    'css/bootstrap.css',

    'css/styles.css',

    'js/app.js',

    'js/apv.js'

];

//Cuando se instala el service Worker, solo se ejecuta una vez

self.addEventListener('install', e => {

    console.log('Instalado el Service Worker');

    e.waitUntil(

        caches.open(nombreCache)

            .then(cache => {

                console.log('cacheando');

                cache.addAll(archivos)

            })

    )

});

Agregar soporte offline:

Agregamos esto acá

//Evento fetch para descargar archivos estaticos

self.addEventListener('fetch', e => {

    console.log('Fetch...', e);

    e.respondWith(

        caches.match(e.request)

            .then(respuestaCache => {

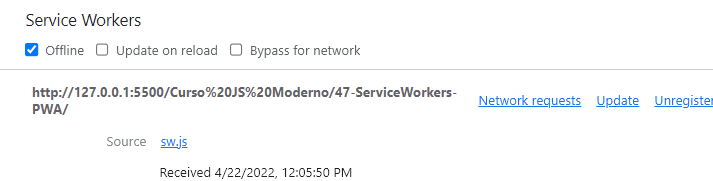
                return respuestaCache

            })

    )

})

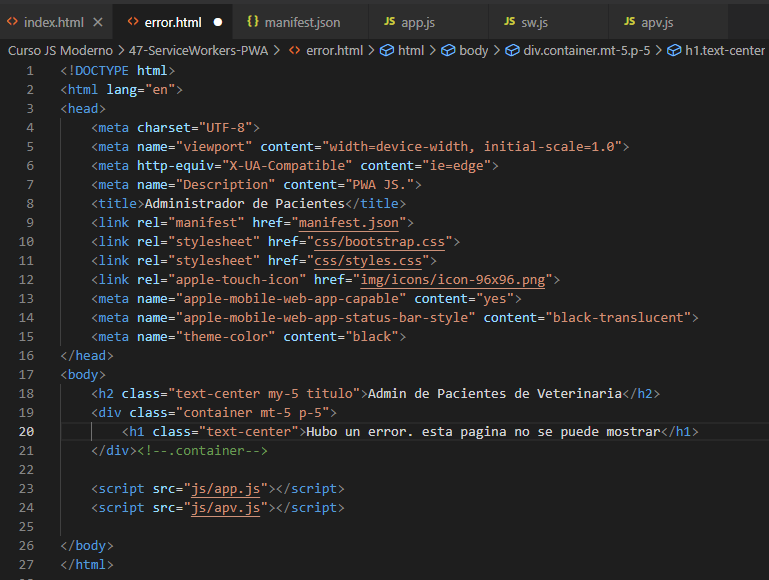
Si le ponemos offlinen y recargamos se tiene que seguir viendo todo



Agregar una pagina de error cuando no hay conexión:

Creamos un archivo error.html

Y dentro le ponemos el contenido del index pero borramos el contenido de los divs, le dejamos todo los link head y demás, y le clavamos un mensaje de error



Para mostrarlo:

Se agrega la pagina de arror al cache y:

const archivos = [

    '/',

    'index.html',

    'css/bootstrap.css',

    'css/styles.css',

    'js/app.js',

    'js/apv.js',

    'error.html'

];

//Evento fetch para descargar archivos estaticos

self.addEventListener('fetch', e => {

    console.log('Fetch...', e);

    e.respondWith(

        caches.match(e.request)

            .then(respuestaCache => {

                return respuestaCache

            })

            .catch( () => caches.match('/error.html'))

    )

})

Crear una nueva versión:

Cambiamos el nombre:

const nombreCache = 'apc-v2';

y esto

//activar serviceWorker

self.addEventListener('activate', e => {

    console.log('Service Worker Activado');

    e.waitUntil(

        caches.keys()

            .then(keys => {

                return Promise.all(

                    keys.filter( key => key !== nombreCache)

                        .map( key => caches.delete(key)) // borra las otras versiones

                )

            })

    )

})

No me funcionó

**Design patterns, patrones de diseño:**

Cada patron es como un plano que se puede personalizar para resolver problemas de diseño en el código.

Pueden ser:

* De creación – Permiten crear objetos y permite a la reutilización del código
* Estructura – Explican como deben comunicarse los objetos y clases en grandes proyectos
* Comportamiento – Se encargan de como se comportan y comunican los objetos

**Class patern:**

ES usar clases

// Class pattern

class Persona{

    constructor(nombre, email){

        this.nombre = nombre;

        this.email = email;

    }

}

const persona = new Persona('Juan', 'Lopez');

console.log(persona);

**Constructor pattern:**

Se usa una clase base para que las otras clases hereden de esta

//constructor pattern

class Persona{

    constructor(nombre, email){

        this.nombre = nombre;

        this.email = email;

    }

}

class Cliente extends Persona {

    constructor(nombre,email, empresa){

        super(nombre,email);

        this.empresa = empresa;

    }

}

const Persona = new Persona('Juan', 'correo@gmail.com'); // se puede instanciar en js

const cliente = new Cliente('Miguel', 'correo', 'jajaja');

**Singleton:**

No permite crear múltiples instancias de una misma clase, pero siempre retorna el objeto instanciado

//Singleton

let instancia = null;

class Persona{

    constructor(nombre, email){

        if(!instancia){

            this.nombre = nombre;

            this.email = email;

            instancia = this;

        }else{

            return instancia;

        }

    }

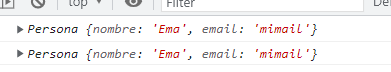
}

const persona = new Persona('Ema', 'mimail');

const persona2 = new Persona('Ema2', 'mimail2'); // aca por mas que se intente definir de nuevo va a retornar siempre la primera instancia que se creo

console.log(persona)

console.log(persona2)



**Factory:**

Es una forma de crear objetos basados en cierta condición, comparten algunos atributos pero en base a las condiciones algunos se reutilizan y algunos son diferentes

//Factory - Crea objetos basados en ciertas condiciones

class InputHTML{

    constructor(type , nombre){

        this.type = type;

        this.nombre = nombre;

    }

    crearInput(){

        return `<input type="${this.type} name="${this.nombre}" id="${this.nombre}">`

    }

}

class HTMLFactory{

    crearElemento(tipo,nombre){

        switch(tipo){

            case 'text':

                return new InputHTML('text', nombre)

            case 'tel':

                return new InputHTML('tel', nombre)

            case 'email':

                return new InputHTML('email', nombre)

            default:

                return;

        }

    }

}

const elemento = new HTMLFactory();

const inputText = elemento.crearElemento('text', 'nombre-cliente')

console.log(inputText.crearInput());

const elemento2 = new HTMLFactory();

const inputText2 = elemento.crearElemento('tel', 'telefono-cliente')

console.log(inputText2.crearInput());

const elemento3 = new HTMLFactory();

const inputText3 = elemento.crearElemento('email', 'email-cliente')

console.log(inputText3.crearInput());

**Module:**

De la manera vieja se hacia así:

//Module Pattern

const modulo1 = (function() {

    const nombre = 'Juan';

    function hola(){

        console.log('Hola')

    }

    return {

        nombre,

        hola

    }

})();

Para poder llamar nombre de otro archivo se hace:

Console.log(modulo1.nombre)

Modulo1.hola();

Con los export es mas fácil igual

**Mixin pattern:**

Es una forma de agregar funciones a una clase una vez que fue creada.

class Persona{

    constructor(nombre, email){

        this.nombre = nombre;

        this.email = email;

    }

}

class Cliente {

    constructor(nombre, email){

        this.nombre = nombre;

        this.email = email;

    }

}

const funcionesPersona = {

    mostrarInformacion() {

        console.log(`Nombre persona: ${this.nombre} email: ${this.email}`);

    },

    mostrarNombre(){

        console.log(`Mi nombre es ${this.nombre}`)

    }

}

//Añadir funcionesPersona a la clase de Persona

Object.assign(Persona.prototype, funcionesPersona);

Object.assign(Cliente.prototype, funcionesPersona);

const cliente = new Persona('Juan', 'correo@');

cliente.mostrarInformacion();

cliente.mostrarNombre();

const cliente2 = new Cliente('Cleinte', 'cliente@mail.com');

cliente2.mostrarInformacion();

cliente2.mostrarNombre();

**Namespace:**

La idea es crear un objeto global y agregar todas las funciones dentro

const restaurantApp = {};

restaurantApp.platillos = [

    {

        platillo: 'Pizza',

        precio: 25

    },

    {

        platillo: 'Hamburguesa',

        precio: 20

    },

    {

        platillo: 'Pancho',

        precio: 20

    }

];

restaurantApp.funciones = {

    mostrarMenu: platillos => {

        console.log(`Bienvenidos a nuestro menu`);

        platillos.forEach((platillo,index) => {

            console.log(`${index} : ${platillo.platillo} $${platillo.precio}`);

        });

    },

    ordenar: id => {

        console.log(`Tu platillo: ${restaurantApp.platillos[id].platillo} se esta preparando`)

    },

    agregarPlatillo: (platillo, precio) => {

        const nuevo = {

            platillo,

            precio

        };

        restaurantApp.platillos.push(nuevo);

    }

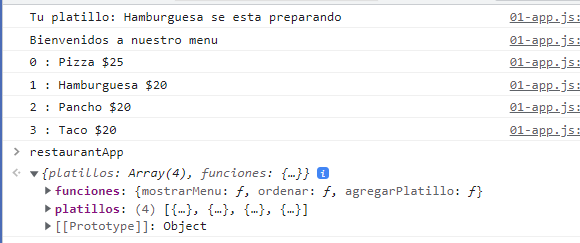
}

restaurantApp.funciones.ordenar( 1 );

restaurantApp.funciones.agregarPlatillo('Taco' , 20);

const { platillos } = restaurantApp;

restaurantApp.funciones.mostrarMenu( platillos );



**Mediador:**

Se comunica con distintos objetos a la vez

function Vendedor(nombre){

    this.nombre = nombre;

    this.sala = null;

}

Vendedor.prototype = {

    oferta: (articulo, precio) => {

        console.log(`Tenemos el siguiente articulo ${articulo}, iniciamos con un precio de ${precio}`)

    },

    vendido: (comprador) => {

        console.log(`Vendido a ${comprador}`);

    }

}

function Comprador(nombre){

    this.nombre = nombre;

    this.sala = null;

}

Comprador.prototype = {

    oferta: (cantidad, comprador)=>{

        console.log(`${comprador.nombre} : ${cantidad}`);

    }

}

function Subasta() {

    let compradores = {};

    return {

        registrar: usuario => {

            compradores[usuario.nombre] = usuario;

            usuario.sala = this;

        }

    }

}

const juan = new Comprador('Juan');

const pablo = new Comprador('Pablo');

const vendedor = new Vendedor('Vendedor de autos');

const subasta = new Subasta();

//registrarlos - subASTA es el mediador que comunica vendedor con compradores

subasta.registrar(juan);

subasta.registrar(pablo);

subasta.registrar(vendedor);

// ----

vendedor.oferta('Mustang 66', 300);

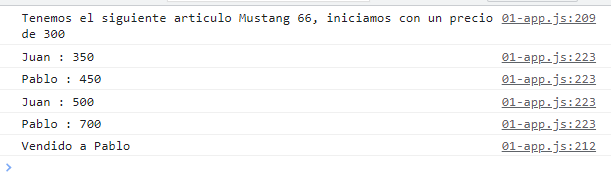
juan.oferta(350, juan);

pablo.oferta(450, pablo);

juan.oferta(500, juan);

pablo.oferta(700, pablo);

vendedor.vendido('Pablo');



**Seguridad-performance-debug:**

**Cuanto tarda en ejecutarse el código?**

Si por ejemplo, queremos hacer una consulta a una api, cuando apretamos el botón podemos ver en la consola, en network, el tiempo que demoro,

También hay una función para saberlo en código con el performance.now(), pones uno antes del código y otro después del código y los restas:

    const inicio = performance.now();

    criptomonedas.forEach( cripto => {

        const { FullName, Name } = cripto.CoinInfo;

        const option = document.createElement('option');

        option.value = Name;

        option.textContent = FullName;

        // insertar el HTML

        criptomonedasSelect.appendChild(option);

    });

    const fin = performance.now();

    console.log(fin - inicio);

el for comparado con el forEach suele ser un toque mas rápido.

**Como ganar algo de performance? Async o defer?**

<script src="js/app.js" async></script>

Con async descarga y ejecuta el código inmediatamente que ha sido descargado, es útil cuando en el js no tenemos algo que modifique al html.

La otra opción es defer, y este no va a ejecutar el código del script hasta que el html este listo, también gana un poco en performance.

<script src="js/app.js" defer></script>

**Debugger:**

Lo que ya se.

**Ofuscar código y ocultarlo:**

Se puede usar esta pagina:

<https://javascript2img.com/>

Hace el código menos legible. Lo que te transforma en la pagina se puede poner en el archivo, reemplazar lo otro, va a seguir funcionando. El tema con esto que así como se puede encriptar se puede desencriptar.

<https://beautifier.io/>

**Otras medidas de seguridad:**

* No almacenar passwords en localStorage ni indexDB
* El DOM Scripting ya escapa los datos y evita riesgos de seguridad, utilizar textContent la mayoría de las veces
* innerHTML, usarlo solo cuando la fuente de los datos es segura. Si estas usando tu propia API si lo podes usar.
* Validar los formularios en el cliente(javascript) para retroalimentación en tiempo real, pero también en el servidor con node o cualquier framework.
* Si se desea crear apps con autenticación de usuarios usar JWT o la herramienta de Auth0
* Cuando se trabaje con dependencias, utiliza una herramienta para verificar vulnerabilidades como **snyk.io**
* Ofuscar el código si es necesario
* Hashear información sensible con bcrypt

**TESTING:**

Ta bueno para saber si una función que hiciste no rompió algo mas.

**Tipos de testing:**

* End-to-end: Mas interactivo, simula algunos clicks , llenar formularios y asegurarse de que se muestre en pantalla lo que se desea. Es el mas usado cypress
* Integracion – Revisar que múltiples partes de nuestro proyecto funcionen bien.
* Unit – Revisar que cada parte por si sola funcione bien
* Static – Revisar por errores en el código mientras vas escribiendo

**Herramientas para testing:**

* Jest – Hay versiones para VueJS, Angular, TypeScript, Node, React, etc. Es necesario tener instalado Node.Js es la mas popular
* Cypress – es mas enfocada para hacer testing end to end

**Testing sin herramientas:**

Con console.logs:

//Probar 2 valores

function suma(a,b){

    return a+b;

}

function restar (a,b){

    return a - b;

}

let resultado = suma(1,2);

let esperado = 3;

if(resultado != esperado){

    console.error(`El ${resultado} es diferente a lo esperado`);

}else{

    console.log('La prueba paso');

}

resultado = restar(10,5);

esperado = 5;

if(resultado != esperado){

    console.error(`El ${resultado} es diferente a lo esperado`);

}else{

    console.log('La prueba paso');

}

**Expected y toBe:**

En expected va el valor esperado y en toBe el resultado.

function suma(a,b){

    return a+b;

}

function restar (a,b){

    return a - b;

}

let resultado = suma(1,2);

let esperado = 3;

expected(esperado).toBe(resultado);

resultado = restar(10,5);

esperado = 5;

expected(esperado).toBe(resultado);

function expected(esperado ){

    return {

        toBe(resultado){

            if(resultado != esperado){

                console.error(`El ${resultado} es diferente a lo esperado`);

            }else{

                console.log('La prueba paso');

            }

        }

    }

}

También podes agregar otras funciones:

function expected(esperado ){

    return {

        toBe(resultado){

            if(resultado != esperado){

                console.error(`El ${resultado} es diferente a lo esperado`);

            }else{

                console.log('La prueba paso');

            }

        },

        toEqual(resultado){

            if(resultado != esperado){

                console.error(`El ${resultado} no es igual a lo esperado`);

            }else{

                console.log('La prueba paso');

            }

        }

    }

}

**Testing con JEST:**

Primero hay que crear un archivo llamado package.json , JEST funciona con NodeJS . Entonces en la terminal pones

**Npm init**

Podes poner una descripción , podes poner un git si queres, pones el autor y darle a yes.

Jest es para desarrollo no vas a hacer los test en producción.

Después haces **npm i –save-dev jest** para que lo guarde como dependencia de desarrollo

Con el comando anterior además de instalar jest crea la carpeta node modules y el package.lock

El package-lock nunca hay que modificarlo.

En la carpeta node-modules y bin hay una serie de archivos binarios que podes ejecutar, dentro de esos esta JEST, entonces en el package.json podemos modificar y hacer esto:

Ponerle en test: jest esto va a hacer que se ejecute el binario de jest



Para llamar al script de test se puede poner en consola

**Npm run test o npm test o npm t**

En la carpeta padre ahora hay que crear una carpeta que sea para los test , la llamamos \_\_tests\_\_ ese nombre jest lo puede comprender

Ahí podemos crear un archivo js donde se pueden hacer las pruebas. Podes poner algo así como

test('Hola mundo con Jest', () => {});

y después corres el npm test en consola

Si no se quiere hacer una carpeta aparte para guardar todos los test se puede poner en el main pero hay que ponerle al nombre del archivo .test.js ej. Hola.test.js

Poner **test** o **it** es lo mismo en el código

Si queres juntar varios test lo podes hacer dentro del describe algo así:

const password = "123456";

describe('Valida que el password no este vacio y tenga una extension de 6 caracteres', () =>{

    test('Que el password tenga 6 caracteres' , () =>{

        expect( password ).toHaveLength(6);

    })

    test('password no vacio', () =>{

        expect(password).not.toHaveLength(0);

    })

})

En el expect le pasamos lo que validamos

toBe : es que lo que pongas en expect sea igual a lo que pones en toBe

Con el not negas

**toHaveLength(numero):** Decis la cantidad de caracteres que queres que tenga el string

Se puede hacer también con arrays:

const carrito = ['Producto 1', 'Producto 2', 'Producto 3'];

describe('Testing a carrito de compras', ()=>{

    test('Probar que el arrat yenga 3 elementos', ()=>{

        expect(carrito).toHaveLength(3);

    })

    test('Verificar que no este vacio', ()=>{

        expect(carrito).not.toHaveLength(0);

    })

})

**Hacer testing a un objeto:**

const cliente = {

    nombre: 'Ema',

    balance: 500

}

describe('Testing al cliente', () =>{

    test('El cliente es premium', () =>{

        expect(cliente.balance).toBeGreaterThan(400); // preguntamos si el balance es mayor a 400

    })

    test('Es Ema el cliente', ()=>{

        expect(cliente.nombre).toBe('Ema'); // saber si el cliente es ema

    })

    test('No es otro cliente', ()=>{

        expect(cliente.nombre).not.toBe('Pedro'); //Saber si el cliente no es JuanPablo

    })

    test('No tiene 500', ()=>{

        expect(cliente.balance).not.toBe('500'); // esta va a fallar xq si tiene 500

    })

});

**Hacer testing a una función:**

function suma(a , b){

    return a + b;

}

function restar(a , b){

    return a - b;

}

describe('Test a las funciones suma y resta', ()=>{

    test('suma de 20 y 30', ()=>{

        expect(suma(20 , 30)).toBe(50);

    })

    test('resta de 10 y 5', ()=>{

        expect(restar(10,5)).toBe(5);

    })

    test('Que la suma 10 y 10, no sea 10', () =>{

        expect(suma(10,10)).not.toBe(10);

    })

    test('Que la reta 10 y 5, no sea otro valor', () =>{

        expect(restar(10,5)).not.toBe(2);

    })

})

**Snapshots:**

Son datos que se almacenan en un string y sobre ella podemos comparar si el objeto es el mismo u otro

Podemos hacer esto

const cliente = {

    nombre: 'Ema',

    balance: 500,

    tipo: 'Premium'

}

describe('Testing al cliente', ()=>{

    test('Es Ema', () =>{

        expect(cliente).toMatchSnapshot();

    })

})

Con el toMatchSnapshot() podes correr después el npm t y te va a crear una carpeta llamada \_\_snapshots\_\_ en la que va a haber un string con los datos del objeto

Después si le mandas otro nombre por ejemplo en vez de Ema que sea Ema 2 te va a fallar el test xq no coincide con el snapshot,

El snapshot se puede modificar desde el mismo archivo, borrarlo y volverlo a crear, o hacer en consola npm t -- -u para actualizar el snapchot con lo que hay en el archivo.

**Agregar babel para realizar pruebas de funciones en otros archivos**

Cuando queres exportar una función de otro archivo, o importarlo, en NodeJS el import no existe, entonces JEST tampoco sabe que hacer con el import, refiriéndonos a esto:

import {suma} from '../js/funciones.js';

Ahí normalmente importaríamos la función suma de otro archivo.

La opción es usar babel, que te permite escribir código de versiones nuevas de js y llevarlo a una versión transpilada anterior, por ejemplo si tenes internet explorer y le tenes que dar soporte esto te sirve, xq creo que no soporta las arrow functions x ej.

Lo uqe hacemos es en el projecto crear un archivo que se llame .babelrc y hay que instalar una dependencia que se llama babel preset entonces ponemos npm i --save-dev @babel/preset-env

En el archivo babel hacemos:

{

    "presets": [

        [

            "@babel/preset-env", {

                "targets": {

                    "node": "current"

                }

            }

        ]

    ]

}

Esto nos va a permitir hacer los imports

**Testing con Cypress:**

Es el que damejores resultados, ofrece pruebas que antes no se podían hacer. Con cypress es posible saber que ve el usuario en pantalla. Cypress.io es la pagina

Tiene pruebas end-to-end pero también unitarias y de integración.

Hacemos npm init

Después

Npm i –save-dev cypress

En estecaso usamos npx puede ser

Npx cypress open te abre la app de escritorio

Npx cypress run te corre en el cli

En la carpeta que se creo de instances, ahí podemos crear nuestros test, tienen que terminar los nombres en .spec.js

Solo se puede usar it() no se puede usar test() en cypress

Para que no te joda el autocompletado podes poner arriba del archivo esta línea: con las 3 barras y todo

/// <reference types="cypress"/>

Lo primero que hay que indicarle es que pagina es en donde va a hacer el test con visit y después en este caso el contains busca si el h1 tiene ese texto, el should exist dice que debería existir un h1.

/// <reference types="cypress"/>

describe('Carga la pagina principal', ()=>{

    it('Carga la pagina principal', ()=>{

        cy.visit('http://127.0.0.1:5500/52-Testing-Cypress/index.html');

        cy.contains('h1', 'Administrador de Pacientes de Veterinaria');

        cy.get('h1').should('exist');

    })

})

El get no se debería usar con selectores generales, se debería hacer así:

En el html se pone un atributo data-cy con un nombre para llamarlo

<h1 data-cy="titulo-proyecto" class="text-center my-5 titulo">Administrador de Pacientes de Veterinaria</h1>

Y en el js del test lo llamamos así:

cy.get('[data-cy="titulo-proyecto"]').should('exist');

Una manera mejor de verificar si el elemento existe y cual es su contenido es así:

        //verificar que el elemnto existe y contenga un texto especifico

        cy.get('[data-cy="titulo-proyecto"]')

            .invoke('text')

            .should('equal', 'Administrador de Pacientes de Veterinaria')

En vez de equal podes usar not.equal para negar

**Validar formularios con cypress:**

/// <reference types="cypress"/>

describe('Validar formulario', ()=>{

    it('Submit y mensaje de error', ()=>{

        cy.visit('http://127.0.0.1:5500/52-Testing-Cypress/index.html');

        cy.get('[data-cy="formulario"]')

            .submit(); //para que se presione el boton de enviar

    })

})

Si queremos validar el mensaje de error cuando no se llenan todos los datos, vamos a tener que agregar el data-cy desde la función de js, que esta función agrega la clase de error:

        //agregar data cy

        divMensaje.dataset.cy = 'alerta';

y el de test quedaría así testeando el mensaje de error:

/// <reference types="cypress"/>

describe('Validar formulario', ()=>{

    it('Submit y mensaje de error', ()=>{

        cy.visit('http://127.0.0.1:5500/52-Testing-Cypress/index.html');

        cy.get('[data-cy="formulario"]')

            .submit(); //para que se presione el boton de enviar

        //Seleccionamos la alerta

        cy.get('[data-cy="alerta"]')

            .invoke('text')

            .should('equal', 'Todos los campos son Obligatorios')

    })

})

Para saber si le agrega la clase success cuando pasa ok ponemos

/// <reference types="cypress"/>

describe('Validar formulario', ()=>{

    it('Submit y mensaje de error', ()=>{

        cy.visit('http://127.0.0.1:5500/52-Testing-Cypress/index.html');

        cy.get('[data-cy="formulario"]')

            .submit(); //para que se presione el boton de enviar

        //Seleccionamos la alerta

        cy.get('[data-cy="alerta"]')

            .invoke('text')

            .should('equal', 'Todos los campos son Obligatorios')

        cy.get('[data-cy="alerta"]')

            .should('have.class', 'alert-danger')

    })

})

**Extra:**

Se puede poner en el archivo cypress.json la url base

{

    "viewportHeight": 1500,

    "viewportWidth": 1200,

    "baseUrl": "http://127.0.0.1:5500/52-Testing-Cypress"

}

Entonces en los test solo ponemos:

cy.visit('/index.html');

**Escribir en formulario con cyress:**

/// <reference types="cypress"/>

describe('Carga la pagina principal', ()=>{

    it('Carga la pagina principal', ()=>{

        cy.visit('/index.html');

        cy.get('[data-cy="mascota-input"]')

            .type('Hook') // con el type tipeamos

        cy.get('[data-cy="propietario-input"]')

            .type('Ema')

        cy.get('[data-cy="telefono-input"]')

            .type('123234')

        cy.get('[data-cy="fecha-input"]')

            .type('2022-10-10')

        cy.get('[data-cy="hora-input"]')

            .type('20:30')

        cy.get('[data-cy="sintomas-textarea"]')

            .type('Solo come')

        cy.get('[data-cy="submit-cita"]')

            .click();

        cy.get('[data-cy=citas-heading]')

            .invoke('text')

            .should('equal', 'Administra tus Citas ')

        cy.get('[data-cy="alerta"]')

            .invoke('text')

            .should('equal', 'Se agregó correctamente')

        cy.get('[data-cy="alerta"]')

            .should('have.class', 'alert-success')

    })

})

**Modificar citas con cypress:**

Le agregamos cy al botón desde js

            //Dataset de ctpress

            btnEditar.dataset.cy= 'btn-editar';

después

/// <reference types="cypress"/>

describe('Llenar campos para una nueva cita y la edita', ()=>{

    it('Campos nueva cita', ()=>{

        cy.visit('/index.html');

        cy.get('[data-cy="mascota-input"]')

            .type('Hook') // con el type tipeamos

        cy.get('[data-cy="propietario-input"]')

            .type('Ema')

        cy.get('[data-cy="telefono-input"]')

            .type('123234')

        cy.get('[data-cy="fecha-input"]')

            .type('2022-10-10')

        cy.get('[data-cy="hora-input"]')

            .type('20:30')

        cy.get('[data-cy="sintomas-textarea"]')

            .type('Solo come')

        cy.get('[data-cy="submit-cita"]')

            .click();

        cy.get('[data-cy=citas-heading]')

            .invoke('text')

            .should('equal', 'Administra tus Citas ')

        cy.get('[data-cy="alerta"]')

            .invoke('text')

            .should('equal', 'Se agregó correctamente')

        cy.get('[data-cy="alerta"]')

            .should('have.class', 'alert-success')

    });

    it('Edita la cita', ()=>{

        cy.get('[data-cy="btn-editar"]')

            .click()

        cy.get('[data-cy="mascota-input"]')

            .clear() //limpia el input

            .type('Nuevo Hook')

        cy.get('[data-cy="submit-cita"]')

            .click();

        cy.get('[data-cy="alerta"]')

            .invoke('text')

            .should('equal', 'Guardado Correctamente')

        cy.get('[data-cy="alerta"]')

            .should('have.class', 'alert-success')

    })

})

**Otros:**

Si queres crear una variable o constante con un elemento pero noencuentra nada, ósea te retorna null podes usar || para definirle otra cosa

acá si no traenada del local storage le asigna un array vacio

tweets = JSON.parse(localStorage.getItem('tweets')) || [];

**Recargar pagina:**

window.location.reload();

**Convertir string a number:**

    Number(presupuestoUsuario);

**Ir a un html:**

window.location.href = 'index.html';

**Ver los parametros de una URL:**

new URLSearchParams(window.location.search)

-----------------------------------------------------------------------

const parametrosURL = new URLSearchParams(window.location.search);

const idCliente = parseInt(parametrosURL.get('id')); //Saca el parametro id de la url

**Confirmar:**

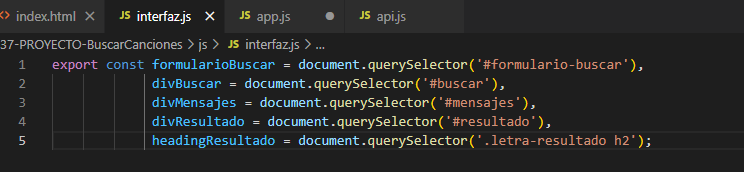
Te devuelve true si da ok o false si da a cancelar

confirm('deseas eliminar este cliente?');

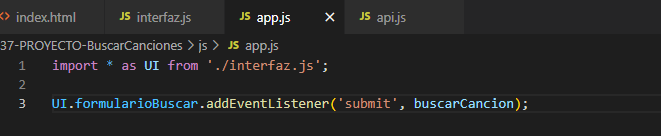
**Importar todas las variables de un modulo:**

Si estas trabajanod con módulos y queres importar todas las variables haces así:

Imaginate que exportas todas estas variables y queres importar todas.



En el app.js haces con el \* la llamada de todos, después podes acceder a cada uno con UI.nombre de la variable



**Para abrir un link en otra pagina de manera segura:**

Para que se abra en otra pagina se usa target=”\_blank” y para que sea seguro se usa rel=”noopener noreferrer”;

<a href="${largeImageURL}" target"\_blank"rel="noopener noreferrer">Ver Imagen</a>

**Alternativa para validar formularios:**

Se crea un objeto con los elementos que se extraen:

        const nombre = document.querySelector('#nombre').ariaValueMax;

        const email = document.querySelector('#email').ariaValueMax;

        const telefono = document.querySelector('#telefono').ariaValueMax;

        const empresa = document.querySelector('#empresa').ariaValueMax;

        const cliente = {

            nombre,

            email,

            telefono,

            empresa

        }

Con esta línea decimos que si hay alguno de los elementos que este vacio te trae false

Object.values(cliente).every(input => input !== '')

Con el every recorre cada uno de los elementos del objeto

Entonces quedaría así:

    function validarCliente(e){

        e.preventDefault();

        const nombre = document.querySelector('#nombre').value;

        const email = document.querySelector('#email').value;

        const telefono = document.querySelector('#telefono').value;

        const empresa = document.querySelector('#empresa').value;

        const cliente = {

            nombre,

            email,

            telefono,

            empresa

        }

        if(validar(cliente)){

            console.log('Todos los campos son obligatorios');

            return;

        }

        console.log('Si se paso la validacion');

    }

    function validar(obj){

        return !Object.values(obj).every(input => input !== '');

    }

**JSON-SERVER:**

**Instalar json-server:**

Json-server permite simular la creación de una api

En el powershell ejecutar como admin, correr npm install -g json-server

Para correrlo es con:

npx json-server --watch db.json no lo probe pero después pone --p 4000 para asignar el puerto

**Otra forma de validar campos:**

function guardarCliente() {

    const mesa = document.querySelector('#mesa').value;

    const hora = document.querySelector('#hora').value;

    //Revisar si hay campos vacios

    const camposVacios = [ mesa, hora].some( campo => campo === ''); //convertimos en arreglo y con some vemos si alguno esta vacio

    if(camposVacios) {

        console.log('Si hay al menos un campo vacio');

    }else{

        console.log('Todos los campos estan llenos');

    }

}

**Traer datos específicos de un objeto:**

const categorias = {

    1: 'Comidas',

    2: 'Bebidas',

    3: 'Postres'

}

Acá el platillo.categoria trae números del 1 al 3

        const categoria = document.createElement('div');

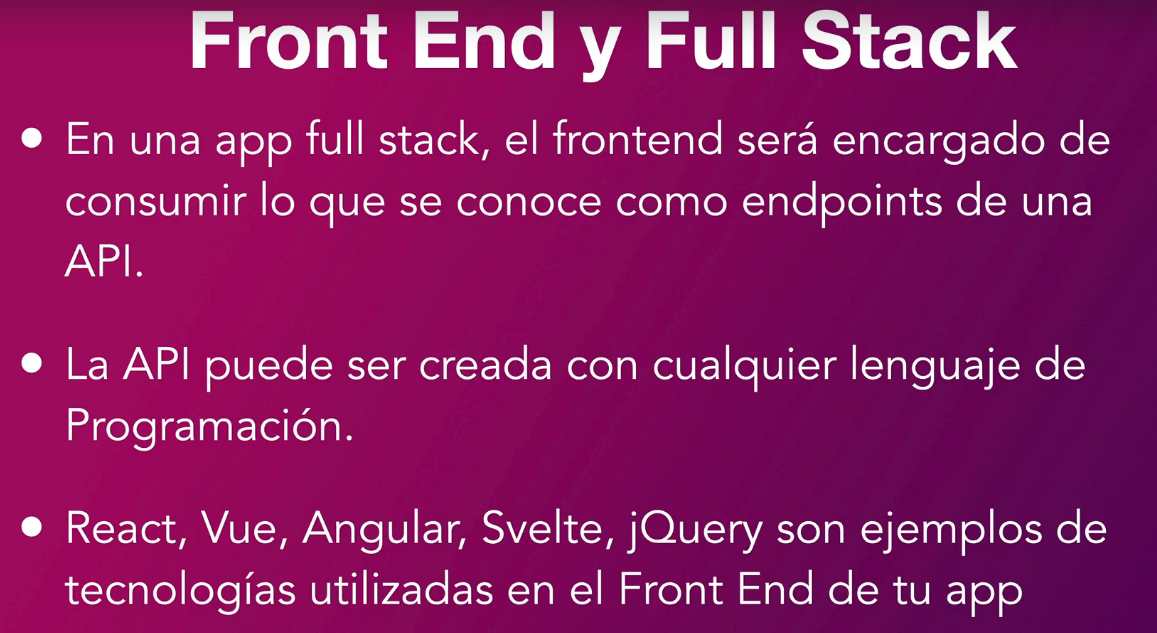
        categoria.classList.add('col-md-3');

        categoria.textContent = categorias[platillo.categoria];

**FULLSTACK Y JAVASCRIPT**

****

****

****

****

****