**CURSO DE JAVASCRIPT DE 0 A PRO**

ECMAScript (ES)

- Especificación para crear un lenguaje de scripting
- Los navegadores lo usan para interpretar JS
- Dicta las reglas, lineamientos qué un lenguaje de scripting debe seguir

JavaScript

- Es un lenguaje de scripting qué respeta las normas de ES
- Es el lenguaje en el qué se programa, ES las reglas qué tiene qué seguir.

Para enlazar un código dentro del HTML hacemos

```
<script src = 'destino del js' >

</script>
```

**Consola de Chrome**

Se accede con inspeccionar.

```
console.log(lo qué quieras mandar a la consola)
```

Si queres mostrar qué algo salió mal en consola, para depuración capaz esté bueno

```
console.error(Mensaje de error)
```

**Creación de variables**

Hay 3 formas

```
var variable = valor
let variable = valor
const variable = valor
```

var es la versión más “vieja”.
Cuando creamos con var, es de tipado dinamico. Si no la inicializamos con ningun valor va a ser undefined. Después tomará el tipo del primer valor qué asignemos.

con let podemos inicializar varias variables de una, separando con ,

```
let variable1,
      variable2,
            variable3
```

*Con var podemos redeclarar la variable, con let no*

**Cadenas de texto**
Se crean con comillas simple generalmente

```
let msj = 'hola qué onda';
```

Para concatenar usamos +

Tenemos el método .length para la longitud

```
msj.length;
```

Método concat para concatenar

```
msj.concat('','hola');
```

Métodos para transformar

```
msj.toUpperCase();
```

```
msj.toLowerCase();
```

Métodos para buscar en cadenas

```
msj.indexOf('texto a buscar') //devuelve la posicion-1
//si no encuentra nada devuelve -1
```

```
msj.substring(posinicial,posfinal) //devuelve subcadenas
```

```
msj.includes('palabra a buscar') //retorna True or False
```

Método para cortar cadenas

```
mensaje.split(' ') //devuelve el mensaje en subcadenas separadas por espacio. Retorna un arreglo
```

Método para reemplazar

```
msj.replace('textoaReemplazar', 'textoNuevo')
```

**Números y operaciones**

```
const numero1 = 30; //ejemplo
```

```
const numero2 = 20;
```

```
let resultado;
```

**Suma**
+

**Multiplicación**
*

**División**
/

**Módulo**
%

**Potencia**
**

```
Math.pow(numero)
```

**Clase Math para Pi y otros valores**

```
Math.PI;
```

**Redondeo**

```
Math.round(numero);
```

```
Math.floor(numero) //redondea hacia arriba
Math.ceil(numero) //redondea hacia abajo
```

**Raíz cuadrada**

```
Math.sqrt(numero)
```

**Valor absoluto**

```
Math.abs(num)
```

**Número mínimo**

```
Math.min(num, num, num)
```

o máximo

```
Math.max(num, num, num)
```

**Valor aleatorio**

```
Math.random()
```

El orden de operadores es el mismo de la aritmética básica

**Incrementos o decrementos**
++ variable
variable+= número a sumar
--variable
variable -= num a restar

**Tipos de dato**
Los tipos se guardan en el valor no en la variable. Es tipado dinamico.
Pueden ser:

- 20
- 'hola'
- true o false
- null
- undefined

Te podes dar cuenta qué es de la forma

```
typeof variable //retorna un string
//los números son de tipo number
```

**Objetos**

```
variable = {
nombre: 'Danilo',
edad: 23,
}
```

**Fecha**

```
variable = new Date()
```

**Operadores de comparación**
Js es raro en este sentido.
>
<
>=
<=
== no es estricto, se pueden comparar números y strings
=== revisa el valor y el tipo de dato, **más estricto, tratar de usar este**

!=

**Orden de mayor a menor de letras**

A < z

la menor la mayor

```
null == undefined        true
null === undefined      false
```

**String a Numero**

```
let var1 = '50'
Number(var1)
```

Esto retorna un 50

También tenemos

```
parseInt(var1)
parseFloat(var1)
```

NaN → not a number

Para truncar un decimal

```
let dato = 1222.323232
dato.toFixed(decimales qué quiero, ej 2)
```

retorna dato = 1222.32

**Número a String**

```
let cp = 3420

String(cp)

cp.toString()
```

**Template Literals o String Template**

```
const prod1 = 'Pizza',
      precio1 = 20

let html;

html = '<ul>' +
```

```
'<li>Orden: ' + prod1 + '</li>'
'<li>Precio: ' + precio1 + '</li>' +
'</ul>;
```

Con comillas invertidas

Está es mucho más limpia, por ende, la mejor!

```
html = `
<ul>
      <li>Orden: ${prod1}   </li>
      <li>Precio: ${precio1}   </li>
</ul>
`
```

**Arreglos en JS**

Se crean

```
const números = [1, 2, 3, 4]
```

otra forma

```
const palabras = new Array('Enero', 'Febrero', 'Marzo')
```

Métodos

```
meses.length();
Array.isArray(palabras);
meses[3] = 'Abril'
meses.push('Mayo')  //lo pone al final
meses.unshift('Inicio') //Lo pone al inicio
meses.pop('Abril') //Elimina un elemento
meses.splice(posinicial,cuantosborro) //elimina con los 2 parametros
meses.reverse()  //lo invierte
meses.concat(otroArreglo) //concatenar arreglos
meses.sort()    //Ordena alfanumericos
```

Para buscar

```
meses.indexOf('Abril')  //retorna la posición
```

Para ordenar números en un arreglo (un asco en JS)

```
arreglo1.sort(function(x, y){
      return x-y

});
```

**La variable const en objetos y arreglos**
No podemos cambiar todo un arreglo a la vez si está definido con const, pero se pueden agregar valores o modificar de a un valor.

**Objetos**
Pueden ser con const o let
Tienen llave y valor

```
let persona = {
      nombre : 'Danilo',
      profesión : 'Dev',
      diaNac : 19,
      musica : ['Rock', 'Techno']
      hogar : {pais: 'Argentina',
              provincia: 'Corrientes'
              }

};
```

Para acceder es con el operador .

```
console.log(persona.nombre);
```

**Arreglos de Objetos**

```
const autos = [
      {modelo: 'Mustang' , motor: 6.0},
      {modelo: 'Camaro' , motor: 6.8},
      {modelo: 'Challenger' , motor: 5.0},
];
```

```
console.log(autos[0]);
```

**Funciones con JS**

Se hacen para resolver pequeños problemas, no hacerlas muy complejas y grandes

```
function saludar() {
      console.log('Hola Danilo');
```

```
};
saludar();
```

Con parámetros

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

saludar('Danilo')
```

```
function sumar(a, b){
return a + b;

};

let suma = sumar(1,3)
```

Otra sintaxis es

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

**IIFE funciones qué se declaran y llaman al momento**

```
(function() {
console.log('Creando un IIFE');
})();
```

Lo anterior se llama al momento de crearse

**Métodos de propiedad (son métodos de objetos)**

```
const musica= {
      reproducir: function() {
console.log('Reproduciendo musica');
},
pausar: function() {
```

```
console.log('Musica pausada');
}
}

musica.reproducir()
```

**Manejo de errores con Try Catch**
Cuando no queremos qué el error de una función no afecte al funcionamiento del programa

```
try {
algo();
} catch(error) {
      console.log(error);
}
```

Muestra el error pero no detiene la ejecución si no encuentra la función algo()

También podemos usar **finally** que cualquiera de los 2 qué se ejecute, finally lo hace de todas formas

```
try {
algo();
} catch(error) {
      console.log(error);
} finally{
console.log('el programa sigue');
}
```

**Fechas**
Se manejan con objetos Date

```
const fechaHoy = new Date();
```

Esto crea un objeto con la fecha actual

Para crear una fecha específica seria:

```
let navidad2017 = new Date('12-25-2017');
```

El formato es mes-dia-año

Métodos para fechas

Para obtener el mes:

`fecha.getMonth()`

Para obtener el dia:

```
fecha.getDate()
fecha.getDay()
```

Para obtener el año

```
fecha.getFullYear()
```

Para obtener minutos

```
fecha.getHours()
```

Para obtener la hora

```
fecha.getHours()
```

Para modificar valores del Date

```
fecha.setFullYear(2025)
```

Se usan setters y getters.

**Estructuras de control IF IF ELSE ELSEIF**

```
if(edad>=18){
console.log('Podes entrar');
} else{
      console.log('No podes entrar');
};
```

Comprobar si una variable tiene un valor

```
if(puntaje) {
console.log('tiene valor')
} else {
      console.log('VACIO')
}
```

Condicional multiple

```
if(hora>0 && hora<=10) {
        console.log('Buen dia');
} else if(hora<=18) {
        console.log('Buenas tardes');
}
```

**&& AND**
**|| OR**

**Switch**

```
switch( condición ) {

case 'valor':
      accion;
      break;
case 'valor 2':
      accion2;
      break;
case 'valor 3':
      accion3;
      break;
default:
      Caso contrario;
      break;
}
```

**Iteraciones**
**For Loop**

```
for (let i = 0; i < puntoFinal; ++ ) {
//código a ejecutar
console.log(arreglo[i]);

}
```

con continue, se saltea el lugar en el qué estoy

**While y Do While**

```
while( condición ) {
    accion;
    i++;
}
```

Si usamos continue acá, tenemos qué incrementar después de el también.

```
continue;
i++;
```

**Do While**

```
do {

      accion;
      i++;

} while(condicion);
```

Entra al menos una vez si o si.

**forEach, Maps e Iteradores**

El forEach sirve para recorrer sobre elementos iterables

```
let pendientes = [];

pendientes.forEach( function(valor, index){

console.log(`En el lugar ${index} : está el valor ${valor}`);


})
```

**Maps**

Map para recorrer un arreglo de objetos

```
let carrito = [
{cod: 1, precio:8},
{cod: 2, precio:10},
{cod: 3, precio:4},

];

const precioProducto = carrito.map(function(){
      return carrito.producto;
});

console.log(precioProducto);
```

**Tenemos un iterador como el de python**

```
for( let auto in automoviles){
console.log(auto)

}
```

**Window Object**

Es la ventana de navegación en si. Accedemos con la palabra reservada:

```
window;
```

Tiene bocha de información del usuario. Con esto se puede mostrar publicidad según el dispositivo qué uses por ejemplo.

console.log() y alert() son métodos del objeto window

**Scope en Javascript**
Es la visibilidad de un valor en JS dentro de una función, iterador, etc.
Determina el alcance de un variable.

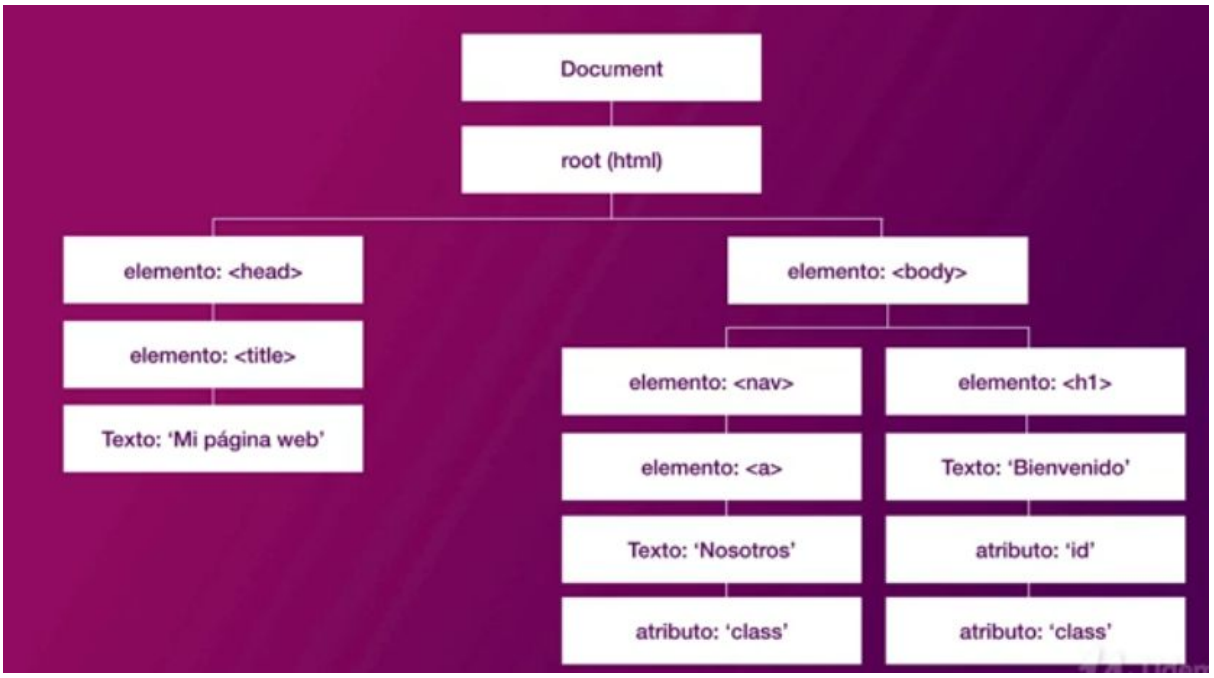
```
let a = 'a';
let b = 'b';
let c = 'c';
```

Así como están las anteriores son variables globales.

Si la ponemos dentro de una función, solo se ven dentro de esa función. La visibilidad de toda la vida rey.

# Qué es el DOM

Document Object
El document object es el padre de todo.




La ventaja del DOM es qué podemos obtener valores del documento con JS y así cambiar esos valores.

**Primeros pasos**
Se trabaja con el ejemplo de Udemy.

```
let elemento;

elemento = document;
```

acá le pasamos el contenido de todo el document a la variable.

Para acceder a las clases tenemos:

```
className;
classList;
```

Con el operador punto ( . ) podemos ir seleccionando cada vez más especificamente una clase o un elemento.

**Seleccionar un elemento y aplicarle propiedades**
Para seleccionar un elemento

```
let elemento;

elemento = document.getElementById('nombre del id');
```

Con text content obtenemos el texto qué está contenido en un elemento

```
.textContent
```

Con style le cambiamos el css a cualquier elemento
ej:

```
.style.background = '#333'
```

**Query Selector**
Se usa igual qué el id selector, pero también podemos seleccionar clases:
Si hay muchas iguales, retorna la primera qué encuentra nada mas

```
document.querySelector('#encabezado')
```

**Seleccionando múltiples elementos**
Se hace con los selectores de clases, después accedemos como si fuera un arreglo.

```
let elemento = document.getElementsByClassName('nombre de la
clase');
```

También podemos con el selector de tag

```
let elementos = document.getElementsByTagName('a')
```

Tenemos también querySelectorAll

```
let elementos = document.querySelectorAll('#principal .enlace');
```

**Traversing**
Es el recorrido de los elementos en cualquier sentido, padre - hijo o hijo - padre

Tipos de enlace con nodeType(retorna un número):

- 1 = Elementos
- 2 = Atributos
- 3 = Text Node
- 8 = Comentarios
- 9 = documentos
- 10 = doctype

Del padre hacia el hijo vamos a ir más especificos con .children las veces qué queramos.

Del hijo hacia el padre seria elemento.parentNode o .parentElement (esté se recomienda mas).

Para recorrido en horizontal tenemos previousElementSibling y nextElementSibling

**Creando elementos con JS**
Se usa ejemplo para añadir un enlace o agregar un texto qué no esté previamente en el html.

```
let enlace = document.createElement('a');

enlace.className = 'enlace';
```

```
enlace.ide = 'nuevo-id'

enlace.setAttribute('href', '#')
```

**Agregar y sacar elementos**

Se hace con los métodos add() y remove()
También tenemos getAttribute() y setAttribute()

Con hasAttribute() comprobamos si tiene o no un atributo.

Con removeAttribute() podemos literalmente eliminar atributos.

**Event Listeners**
Esperar acciones del usuario para ejecutar un codigo.

Se hace con:

```
.addEventListener('evento', función );
```

TODO EN MINÚSCULAS
Los ‘eventos’ del mouse pueden ser:

- click
- dblclick
- mouseenter
- mouseleave
- mouseover
- mouseout
- mousedown (mantener presionado el mouse)
- mouseup
- mousemove

Podemos recibir lo qué retorna un evento se hace con function(e)
e es el evento recibido.
El evento e tiene muchísima información, por ejemplo si el evento fue un click va a

guardar parámetros como en qué parte de la pantalla se clickeo

**Eventos para inputs**

- keydown, cada vez qué se presiona una tecla
- keyup, cada vez qué se suelta una tecla
- keypress
- focus (cuando se selecciona un input)
- blur, cuando se da click afuera, ej validar texto invalido
- cut, para cortar ctrl+x
- copy
- paste
- input
- change
- submit

**Event bubbling**

Es cuando se selecciona un evento y otros son activados o seleccionados.
Por ejemplo si tenemos un boton, dentro de una imagen, dentro de un div. Si clickeamos el boton hasta el último div quizas responda.

Para evitar esto se usa stopPropagation()

**Delegation**

Una forma mejor de evitar el bubbling, no es un reemplazo.
Se hace con .contains() como parámetro le damos la clase por la qué queremos preguntar, usamos un condicional, si es alguno qué no nos interesa cancelamos la acción.

## LOCAL STORAGE

Permite agregar y mostrar datos en el navegador del usuario, no en una base de datos.
Se usa usando localStorage. Si la ventana el navegador o lo qué sea se cierra, la info sigue cargada.
Se agrega algo con:

```
localStorage.setItem('nombre', 'Juan');
```

El formato es ‘llave’ ‘informacion’

También tenemos lo qué esa la sessionStorage qué guarda los datos mientras el usuario sigue en la página.

Podemos eliminar valores:

```
localStorage.removeItem('llave', 'info');
```

Y podemos obtener valores:

```
localStorage.getItem('llave');
```

Para limpiar el localStorage:

```
localStorage.clear();
```

# Proyecto: Do List con local Storage

Tratar de ir enviando siempre a consola los resultados de una función para probar si funciona.
**Si una clase no está en el HTML, es decir, cuando lo agregamos por JS, no le podemos asignar un eventListener. Tenemos qué usar delegation (Primero ponemos event Listener en una clase qué contenga a la clase creada, y después preguntamos por la clase o ID qué hayamos creado.)**

Cuando guardemos cosas en el LocalStorage conviene guardarlo como un arreglo,en vez de:
tweet 1 contenido
tweet 2 contenido
tweet 3 contenido

Nos conviene hacerlo como:
[tweet1, tweet2, tweet3]

EventListener **DOMContentLoaded** carga cuando todo el HTML se haya cargado.

# Proyecto final: Carrito de compras

Para remover elementos de  elemento, ejemplo una tabla, dentro del DOM tenemos 2 formas.

Menos codigo:
elemento.innerHTML = ‘ ’;
return false;
esto sobreescribe con un str vacio entonces lo borra

Y la forma recomendada y más rapida es con un while:

```
while(elemento.firstChild){
        elemento.removeChild(elemento.firstChild);
}
```

esto lo qué hace es mirar si tenemos contenido y mientras tenga algo lo va a borrar.

# Proyecto: Enviar Email

Es buena práctica, cuando queremos ejecutar varias cosas al iniciar la app o la página usar el event Listener DOMContentLoaded con una función de inicioApp()

Validación de campos vacíos

Para mostrar gifs de cargas y cosas así tenemos qué establecer primero en el css display: none.

Y después con JS cambiamos a

img.style.display = ‘block’

Para ejecutar una función con un tiempo determinado

```
    setTimeout(function(){
        funciones();

    }, tiempoAEjecutarenMilisegundos);
```

Cuando se cumpla el tiempo si queremos volver al inicio solo ponemos

elemento.reset()

**OBJETOS**

Para crear un objeto tenemos:

```
const cliente = {
      nombre: 'Capo',
      apellido 'Diez'
};
```

Tenemos la palabra **this.** para acceder a los atributos de un mismo objeto.

Otra forma es creando con un constructor como si fuera una clase

```
function Cliente(nombre, saldo){
      this.nombre = nombre;
      this.saldo = saldo;
      this.tipoCliente = function{}

}

const persona = new Cliente('Capo', 200);
```

Siempre qué creamos algo con new, quiere decir qué es un objeto

**PROTOTIPOS (PROTOTYPES)**
Un objeto en JavaScript tiene otro objeto, llamado prototype (prototipo, en español). Cuando pedimos a un objeto una propiedad que no tiene, la busca en su prototipo. Así, un prototipo es otro objeto que se utiliza como una fuente de propiedades alternativa.
Prototype es una propiedad de Object, -el objeto del que se derivan todos los demás objetos-, y esta propiedad es, a su vez, un objeto. Por tanto, el prototipo último de un objeto es Object.prototype. Este prototipo padre tiene métodos que comparten todos los objetos.

Para crear un prototipo

```
Clase.prototype.funcion()
```

Básicamente, escribimos funciones específicas para los objetos de un tipo. Podemos acceder a todos los atributos y métodos qué tenga ese objeto desde un proto.

**Heredar prototypes a otro Objeto**

Es como la herencia de la POO, se implementa así:

```
function Cliente(nombre,saldo){
      this.nombre = nombre;
      this.saldo = saldo;

};
```

Ahora heredamos para empresa

```
function Empresa(nombre, saldo, telef, tipo){
//aca viene lo importante
Cliente.call(this, nombre, saldo);
this.telef = telef;
this.tipo = tipo;

}
```

Para heredar los métodos del prototype

```
Empresa.prototype = Object.create(Cliente.prototype)
```

**Creando objetos con créate()**

```
const juan = Object.create(clase)
```

Después para darle atributos

```
juan.saldo = 200;
```

**CLASES**

Otra forma de crear clases y objetos es de:

```
class Cliente {
      constructor(atributo1, atributo2){
            this.atributo1 = ''
}


     método(){
        funciones
}


}
```

Atributos estáticos son aquellos qué no se instancian.

```
class Cliente {

static metodoEstatico(){
}
}
```

para llamarlo

```
Cliente.metodoEstatico()
```

**Herencia en classes**

```
class Empresa extends Cliente{
 constructor(atributosDeCliente, atributosNuevos){
//va hacia el constructor padre
super(atributosDeCliente);
      this.AtributosNuevos = valor;
}
}
```

# Proyecto: Cotizador de seguros de autos

Es bueno para ahorrarnos un selector en el HTML con muchas opciones imprimirlas con un bucle en JS.

```
const selectAnios = document.getElementById('anio');

for(let i = max; i > min; i--){
    let option = document.createElement('option');
    option.value = i;
    option.innerHTML = i;
    selectAnios.appendChild(option);


}
```

Esto imprime en una barra el intervalo de años qué va desde el actual hasta 20 antes

Cuando queres leer algo de un option tenes que:

```
    const marcaSeleccionada = marca.options[marca.selectedIndex].value;
```

Para leer datos de un input radio button:

```
    const tipo = document.querySelector('input[name="tipo"]:checked').value;
```

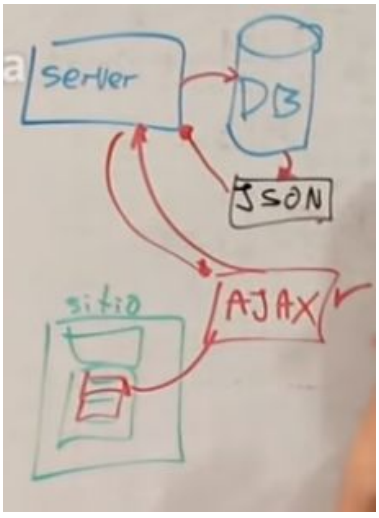
# Proyecto: Gasto Semanal

Para refrescar una ventana lo hacemos con window.location.reload()

**AJAX(asynchronous javascript and xml)**

Es una tecnología para crear tecnología web. Un ejemplo útil es cuando recibimos una notificación en una web, al instante, sin recargar nada.

El cliente le pide los datos a AJAX, a su vez AJAX le pide al servidor la información qué necesito y AJAX modifica la web, el sector qué quiero sin tener qué recargar nada.




Ajax es una opción buena para cargar info en la app sin necesidad de actualizar la página. O para cargar desde una BD, cambiarla, etc. JS no se puede comunicar directamente con archivos externos parece.

Tenemos varias parte para llamar a AJAX:

una parte es el xmlhttprequest qué es un objeto, lo llamamos xhr:

```
const xhr = new XMLHttpRequest();
```

En ese objeto tenemos los métodos para interactuar con AJAX.

Después tenemos qué abrir una conexión, con la acción qué queramos hacer.

```
    //abrir una conexion
    xhr.open('GET', 'datos.txt', true);
```

Los parámetros son, la acción, el link, y la llamada asíncrona.

Después comprobamos el estado en dónde tenemos las siguientes posibilidades.

```
xhr.onload = function(){
    /*ESTADOS: 200 igual a todo correcto
    403 prohibido
    404 no encontrado



    */
}
```

this: hace referencia a la conexión
Para ver los datos qué tenemos se hace:
this.responseText

**La última parte es siempre enviar la conexión:**
xhr.send()

**AJAX CON JSON**
JSON(Java Script Object Notation) objetos de Javascript pero con sintaxis distinta
Es llave-valor, **con comillas dobles**
{
“id”:1,
“nombre”: “Danilo”

}

El último valor no tienen qué llevar coma.

Para varios objetos tenemos qué encerrar entre [ ]

**Acá entra el valor del JSON.parse(), porque convierte un string de JSON en un objeto de JS**

**Siempre qué se quiera hacer una consulta nueva tambien hay qué abrir una conexión nueva**

**API, REST-API Y REQUEST**

- **API**: es una interfaz de aplicación de programación, son las funciones y métodos qué ofrece una librería para ser usada por otro software. Hay qué enviar una petición estructurada para usarla.
- **REST APIS:** REST quiere decir estado representacional de transferencia. Se puede diseñar en cualquier lenguaje y podemos acceder después a ellos. Describe como se ponen a disposición los recursos y datos. Responde a los request de HTTP.
- **Endpoints:** son urls para realizar operación CRUD.
    - GET para obtener
    - POST para crear
    - PUT para editar
    - DELETE para oh sorpresa, borrar

**Leyendo de una API con AJAX**

La mayoría del tiempo vamos a consumir REST apis. Vamos a usar JSON place holder.

# Proyecto: Generador de nombres

Está api ya está descontinuada pero vamos a aprender cómo filtrar.
Tenemos qué ver el formato de la url de la api para poder filtrar las peticiones, en la de nombres se solucionaría con este código:

```
let url = '';
url += `http://uinames.com/api/?`

if(origenSeleccionado!==''){
    url += `region=${origenSeleccionado}&`;
};
if(generoSeleccionado!==''){
    url += `gender=${generoSeleccionado}&`;
};
```

## JAVASCRIPT ASÍNCRONO

- En los programas síncronos, el código se ejecuta secuencialmente. No puede arrancar una línea sin qué la otra haya terminado.
- En el código **asíncrono** las líneas no necesitan esperar a qué las demás terminen de ejecutarse para arrancar.

La mayor utilidad del asincronismo es cuando consultamos APIs para poder seguir trabajando mientras esperamos la respuesta.
Para crear código asíncrono en JS tenemos:

**Callbacks**
Piedra angular de la programación asíncrona.
El forEach es un ejemplo de Inline callback.
Básicamente es una función corriendo dentro de otra.
Podemos también, pasarle una función o varias como parametro a otra función, entonces se van a ejecutar en cadena si es qué una no termina.

**Promises**
Son importantes para trabajar con las Fetch Apis, se usan cuando un valor a revisar capaz no esté disponible en el momento pero en el futuro si. Por ejemplo si estamos leyendo algo de 1000 registros podemos esperar a qué sean todos y después ejecutamos el resolve.

Para crear una: resolve es cuando se cumple la promesa, y el reject se ejecuta cuando no se cumpla la promesa

```
const esperando = new Promise(function(resolve, reject){

setTimeOut(function(){
           resolve('Se ejecuto');


},5000);
//esperamos 5 segundos para enviar el resolve

});
```

Para llamar a un resolve tenemos qué hacer:

```
esperando.then()
```

El .then() espera a qué se cumpla resolve y ahí va a ejecutar el codigo

```
esperando.then(function(mensaje){

      console.log(mensaje);

});
```

Si no creamos el .then() el resolve nunca se va a mandar a llamar.
Generalmente en consola se muestra <pending> cuando no agregamos un then

Ahora agregamos el reject. Por ejemplo lo qué sea qué buscamos no está disponible, entonces con el reject podríamos poner mensajes de error entre otras cosas.

```
const aplicarDescuento = new Promise(function(resolve, reject){
const descuento = true;

if(descuento===true){
resolve('descuento aplicado ')
}else{
reject('No se puede aplicar el descuento');
}



}
});

aplicarDescuento.then(function(resultado){

console.log(resultado);

}).catch(function(error){
console.log(error);
})
```

El reject siempre está sujeto a un .catch()

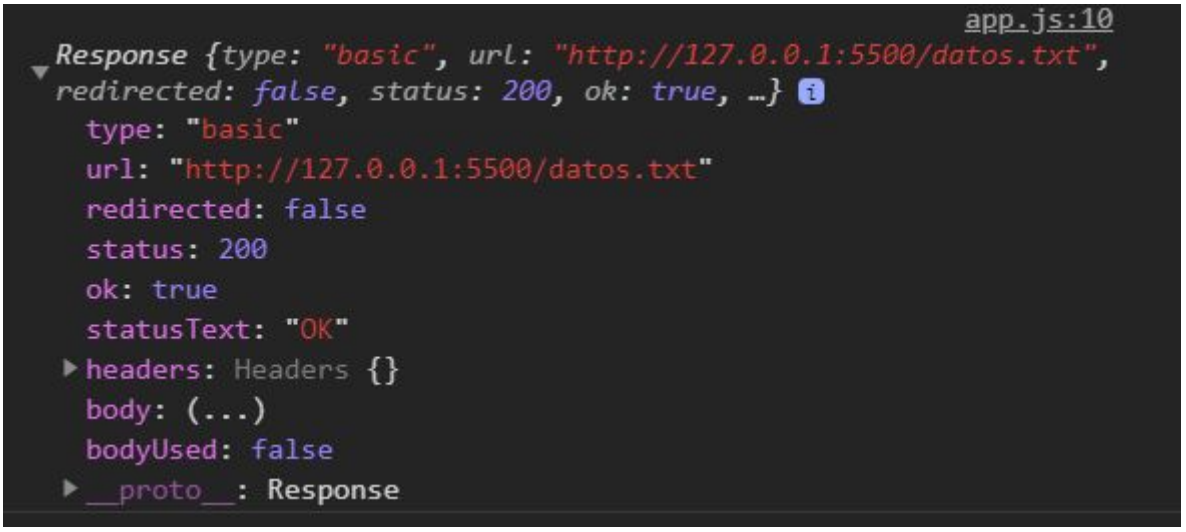
**Fetch API: Tecnología nueva para realizar peticiones.**

Es mucho más fácil de aplicar.

Para cargar un .txt solo tenemos qué hacer:

```
fetch('datos.txt')
```

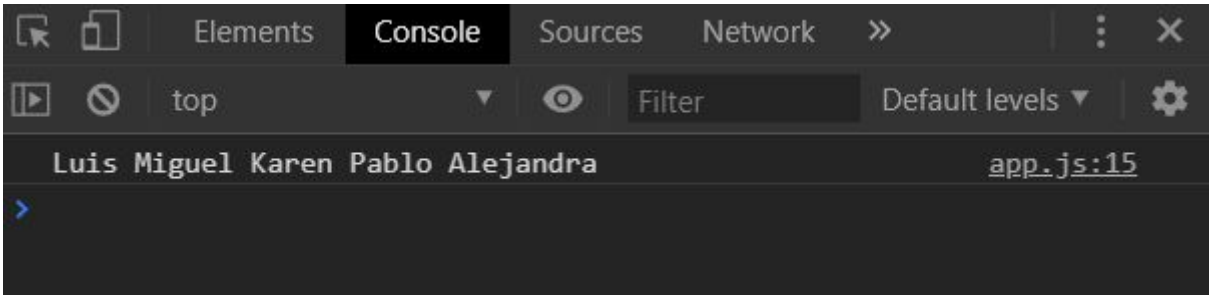
Esto devuelve la ejecución de un Fetch




Generalmente requiere qué usemos 2 .then

El primero para conectarse y traer la infor con el método para ver cómo traernos esa información, y el segundo para leer y acceder como queramos.

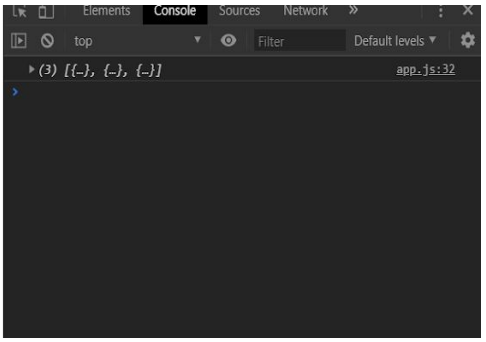
```
function cargarTXT(){

    fetch('datos.txt')
        .then(function(res){

            return res.text();

        })
        .then(function(data){

            console.log(data);

        })
        .catch(function(error){
            console.log(error)
        })

}
```




**Para cargar un .json**

Es casi igual pero tratamos al .json como un arreglo como siempre.

```
function cargarJSON(){

    fetch('empleados.json')
    .then(function(respuesta){
        return respuesta.json();
    })
    .then(function(data){
        console.log(data);
        let html = '';
        data.forEach(function(dato){
            html += `<h4>${dato.nombre}</h4>
            <li>${dato.puesto}</li>
            `



        })

        document.getElementById('resultado').innerHTML = html;
    })
    .catch(function(error){
        console.log(error);

    })

};
```

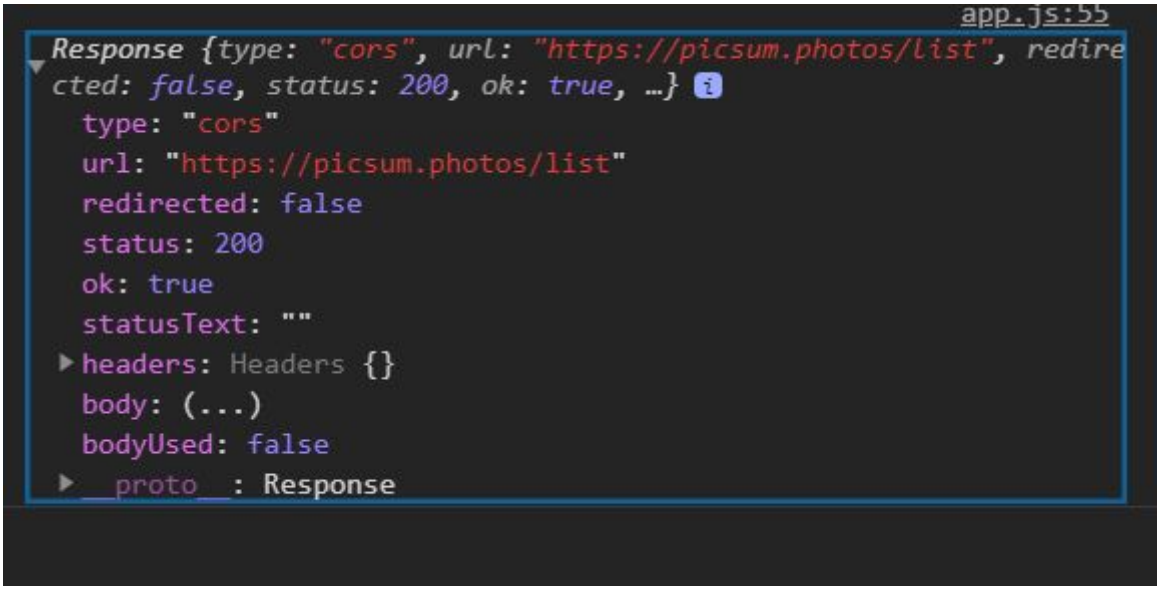




**Para cargar un REST**

Usamos fetch con el enlace

```
fetch('https://picsum.photos/list')
```

Devuelve esto

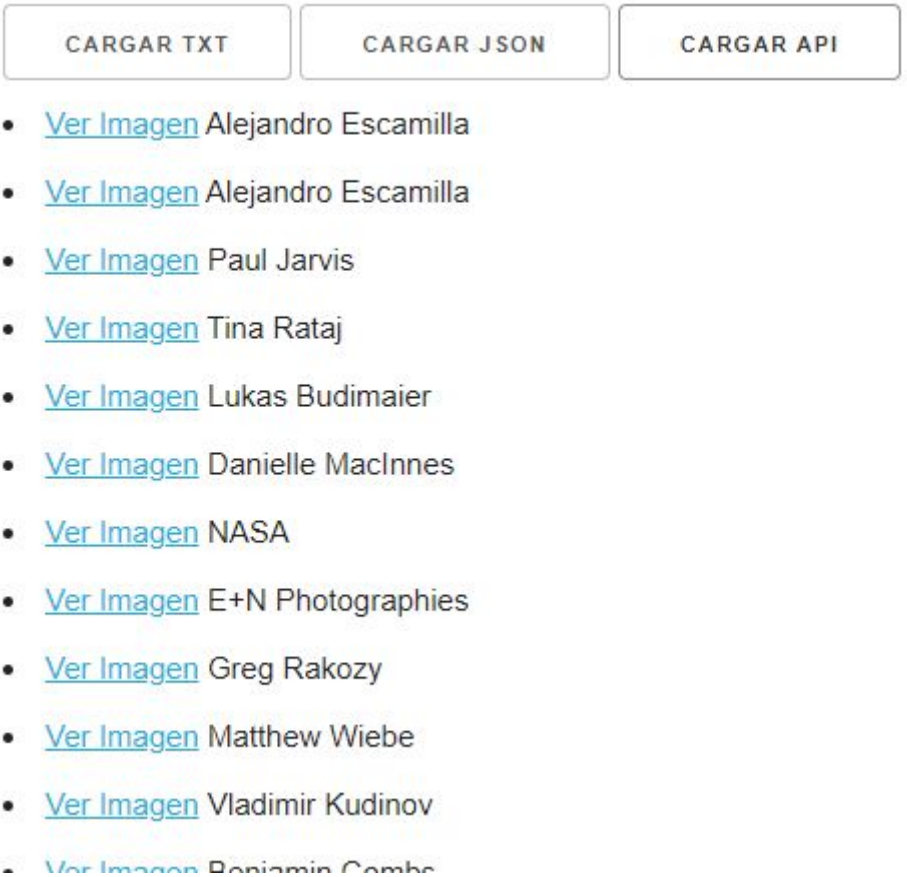



```
function cargarRest(){
    fetch('https://picsum.photos/list')
    .then(function(resp){
        return resp.json();

    })
    .then(function(imagenes){
        console.log(imagenes);

    })
}
```

```
                                                              app.js:59
(993) [{…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…},
{…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…},
{…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…},
{…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…},
{…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…},
{…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…},
{…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…},
{…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…},
{…}, {…}, {…}, {…}, {…}, …] i
 ▶ [0 … 99]
 ▼ [100 … 199]
   ▶ 100: {format: "jpeg", width: 1544, height: 1024, filename:…
   ▶ 101: {format: "jpeg", width: 1500, height: 1000, filename:…
   ▶ 102: {format: "jpeg", width: 3264, height: 2176, filename:…
   ▶ 103: {format: "jpeg", width: 2500, height: 1667, filename:…
   ▶ 104: {format: "jpeg", width: 4928, height: 3264, filename:…
   ▶ 105: {format: "jpeg", width: 1600, height: 1067, filename:…
   ▶ 106: {format: "jpeg", width: 4147, height: 2756, filename:…
   ▶ 107: {format: "jpeg", width: 4928, height: 3264, filename:…
   ▶ 108: {format: "jpeg", width: 3504, height: 2336, filename:…
   ▶ 109: {format: "jpeg", width: 1500, height: 1000, filename:…
   ▶ 110: {format: "jpeg", width: 4272, height: 2511, filename:…
   ▶ 111: {format: "jpeg", width: 4032, height: 2272, filename:…
   ▶ 112: {format: "jpeg", width: 3823, height: 2549, filename:…
   ▶ 113: {format: "jpeg", width: 4910, height: 3252, filename:…
   ▶ 114: {format: "jpeg", width: 2500, height: 1667, filename:…
   ▶ 115: {format: "jpeg", width: 3807, height: 2538, filename:…
   ▶ 116: {format: "jpeg", width: 4698, height: 3166, filename:…
   ▶ 117: {format: "jpeg", width: 1600, height: 1066, filename:…
   ▶ 118: {format: "jpeg", width: 2742, height: 1828, filename:…
   ▶ 119: {format: "jpeg", width: 4928, height: 3264, filename:…
   ▶ 120: {format: "jpeg", width: 2560, height: 1920, filename:…
   ▶ 121: {format: "jpeg", width: 4032, height: 2272, filename:…
   ▶ 122: {format: "jpeg", width: 4752, height: 3168, filename:…
   ▶ 123: {format: "jpeg", width: 3465, height: 3008, filename:…
   ▶ 124: {format: "jpeg", width: 2500, height: 1667, filename:…
   ▶ 125: {format: "jpeg", width: 2448, height: 2448, filename:…
   ▶ 126: {format: "jpeg", width: 2048, height: 1365, filename:…
   ▶ 127: {format: "jpeg", width: 4272, height: 2848, filename:…
   ▶ 128: {format: "jpeg", width: 3600, height: 2385, filename:…
```

```
function cargarRest(){
    fetch('https://picsum.photos/list')
    .then(function(resp){
        return resp.json();
```

```
    })
    .then(function(imagenes){
        console.log(imagenes);
        let html='';
        imagenes.forEach(function(imagen){
            html += `
            <li>
            <a href = "${imagen.post_url}">Ver Imagen</a>
            ${imagen.author}

            </li>


            `

        })
        document.getElementById('resultado').innerHTML = html;
    })
}
```




**ARROW FUNCTIONS**

Una función flecha es una alternativa sintácticamente más compacta que una función convencional. Dichas funciones no realizan sus propias vinculaciones de this, arguments, super o new.target, y no son adecuadas para ser utilizadas como métodos, además de que no pueden ser usadas como constructores.

```
let aprendiendo = () => {

console.log('Hola')

}
aprendiendo();
```

Si dejamos solo un string o un objeto o lo qué sea, no tenemos qué darle ninguna operación, ejemplo:

```
let aprendiendo = ()=>{
```

```
'Hola'
}
console.log(aprendiendo());
```

Para pasar parámetros:

```
let aprendiendo = (mensaje)=>{

console.log(mensaje)

}
aprendiendo('hola');
```

si le pasamos un solo parámetro podemos no poner ()

Sirven para los forEach

```
productos.forEach(producto=> {console.log(producto)

})
```

También son útiles para las promesas

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

**ASYNC / AWAIT**

Sirve para crear funciones asíncronas. No está soportado en todos los navegadores. Una async function siempre requiere un promise. El await espera la ejecución hasta qué el promise se haya ejecutado.

```
async function obtenerClientes(){
    const clientes = new Promise((resolve, reject) =>{
        setTimeout( ()=>{

            resolve(`Clientes descargados...`);


        },2000);
```

```
    });
    const error = true;
    if (error===false){

        const respuesta = await clientes;
        return respuesta;
    }else{

        await Promise.reject(`Hubo un error`);
    }

}

obtenerClientes()
.then(res=>console.log(res))
.catch(error=>console.log(error));
```

EL ASYNC AWAIT SIEMPRE RETORNA UNA PROMESA

**Consumiendo una fetch API con async await**

https://jsonplaceholder.typicode.com/todos

```
async function leerTodos(){
    const respuesta = await
fetch('https://jsonplaceholder.typicode.com/todos');


    //arranca cuando la respuesta este hecha
    const datos = await respuesta.json();
    return datos;


}

leerTodos()
    .then( usuarios => console.log(usuarios) );
```

# Proyecto: Cotizador de criptomonedas

```
cotizador.obtenerMonedasAPI()
.then(monedas =>{
    //esto retorna una arreglo
    for(const[key, value] of Object.entries(monedas.monedas.Data)){
        console.log(key)

    }



})

//lo anterior es un objeto de objetos, no es un arreglo.
//entonces no podemos usar forEach
```

el for está bueno para recorrer objetos, los convierte a arreglos, pero tenemos qué tener si o si el key y el value.

**Asignación por Destructuring**
Es un código qué ayuda a extraer valores de un objeto u arreglo
Sirve para evitar por ejemplo:
${persona.dni} + ${persona.nombre} + ${persona.apellido}

Para resolver esto:

```
const cliente = {
   nombre: "Danilo",
   apellido: "Diez"
}

let {nombre, tipo} = cliente;
```

**Destructuring en arreglos**

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

const [primeraCiudad, segundaCiudad, tercerCiudad] = ciudades;

console.log(primerCiudad);

const [ , , , paris] = ciudades;
```

**Destructuring a funciones**

```
function reservacion(completo, opciones){
   opciones = opciones || {};
/*
   let metodo = opciones.metodoPago,
       cantidad = opciones.cantidad,
       dias = opciones.dias;
*/

let {metodoPago, cantidad, dias} = opciones;
```

```
    };

//
reservacion(
    true,
    {
        metodoPago: 'tarjetaCredito',
        cantidad: 3000,
        dias: 3
    }

);
```

## Symbols

***Symbol*** es un tipo de datos cuyos valores son únicos e immutables. Dichos valores pueden ser utilizados como identificadores (claves) de las propiedades de los objetos.
Permiten crear propiedades unicas. Cada Symbol es diferente a los demás.

```
const simbolo = Symbol();

const simboloConDescripcion = Symbol('Esta es una descripcion');



console.log(simbolo);
console.log(simboloConDescripcion);
```

```
const nombre = Symbol();
let persona = {};

persona.nombre = 'Juan';     //Esto lo agrega como propiedad

persona[nombre] = 'Juan';    //Esto lo agrega como simbolo al objeto
```

Los symbols se consideran como propiedades **privadas** del objeto.

**Sets**

Un set permite crear una lista de valores. Es como un arreglo pero sin duplicados.

```
let carritoCompras = new Set();


console.log(carritoCompras);
```

```
danilodiez@notebook:~/Escritorio$ node app.js
Set {}
```

Para agregar datos usamos .add()

```
let carritoCompras = new Set();
carritoCompras.add('camisa');
console.log(carritoCompras);
```

```
danilodiez@notebook:~/Escritorio$ node app.js
Set { 'camisa' }
```

También podemos inicializar el set pasandole un arreglo

```
let numeros = new Set([1,2,3,4])
console.log(numeros);
```

```
Set { 1, 2, 3, 4 }
```

Para averiguar si un set tiene un determinado dato, tenemos el has()

Retorna true o false

```
console.log(numeros.has(1));
```

Para borrar usamos delete()

```
numeros.delete(1);


console.log(numeros);
```

```
Set { 1, 2, 3, 4 }
Set { 2, 3, 4 }
```

Para iterarlos usamos el forEach() de toda la vida.

Para convertir un set a un Array tenemos lo siguiente. Usamos 3 puntos

```
const arregloNumeros = [...numeros];
```

**Maps**

Son listas ordenadas, pero almacenan la información de la forma llave: valor

```
let cliente = new Map();

console.log(cliente);
```

```
danilodiez@notebook:~/Escritorio$ node app.js
Map {}
```

Para cargar un map tenemos .set()

```
cliente.set('nombre', 'Danilo')
cliente.set('Saldo', 3000)

console.log(cliente);
```

```
danilodiez@notebook:~/Escritorio$ node app.js
Map { 'nombre' => 'Danilo', 'Saldo' => 3000 }
```

Para obtener los datos de un map usamos .get()

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

También tenemos el .has(llave) para preguntar por un valor determinado

Para borrar tenemos .detele(llave)

Para limpiar el mapa tenemos .clear()

Para agregar propiedades por defecto al Map tenemos

```
let cliente = new Map(
   [['nombre', 'paciente'],
   ['camilla', 'no asignada']

]);
```

Para recorrerlo también tenemos el forEach()

**Generadores**
Son funciones qué retornan un iterador.

# Proyecto: mostrar lugares en un mapa

Para cuando quiera hacer algo con mapas tengo el framework **leaflet,** se instancia haciendo:

```
class UI {
    constructor() {

        // Iniciar el mapa
        this.mapa = this.inicializarMapa();

    }

    inicializarMapa() {
        // Inicializar y obtener la propiedad del mapa
        const map = L.map('mapa').setView([19.390519, -99.3739778], 6);
        const enlaceMapa = '<a
href="http://openstreetmap.org">OpenStreetMap</a>';
        L.tileLayer(
            'http://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png', {
            attribution: '&copy; ' + enlaceMapa + ' Contributors',
            maxZoom: 18,
            }).addTo(map);
        return map;

    }
```

**Para crear un popUp**

```
            //crear un popUp
            const opcionesPopUp = L.popup().setContent(
                `
```

```
                <p>
                Calle: ${calle}
                </p>
                <p>
                Regular: $${regular}
                </p>
                <p>
                Premium: $${premium}
                </p>

                `


            )
```

```
            const marker = new L.marker([
                parseFloat(latitude),
                parseFloat(longitude)

            ]
            ).bindPopup(opcionesPopUp);
```

Le agregamos bindPopup al marcador.

### Crear un buscador

```
const buscador = document.querySelector('#buscar input')

buscador.addEventListener('input', ()=>{

    if(buscador.value.length>5){

        ui.Buscar(buscador.value)

    }else{
      ui.mostrarEstablecimientos()
      };

})
```

```
    buscar(busqueda){
        this.api.obtenerDatos()
```

```
        .then(datos =>{
            const resultados = datos.respuestaJSON.results
      this.filtrarSugerencias(resultados, busqueda);
      });




    }

    //filtra las sugerencias
    filtrarSugerencias(resultado, busqueda){
        //filtrar con .filter

        const filtro = resultado.filter(filtro =>
filtro.calle.indexOf(busqueda) !==-1)



        this.mostrarPines(filtro);
    }
}
```

**Módulos**
Funciones del programa agrupados. En un solo archivo. Sirve para organizar el código.
Para importar un módulo tenemos:
<script src="modulo.js"></script>
Cada llamada de módulo es http, así qué va a ser al sitio más lento. Las dependencias son manuales, así qué hay qué ver el orden en el qué hacemos las llamadas.

Otra forma, es creando un solo archivo, pero queda enorme.

**Creando módulos**
Para enviar una variable a otro módulo tenemos export qué se tiene qué combinar con el uso de type = "module" en el llamado del html. Se usa un export para cada

variable, clase o función. Para leerlo en otro módulo usamos import {Lo qué queramos traer} from ‘./nombreArchivo’
Tenemos qué declarar el tipo de módulo en cada archivo para poder usar export e import.
Para traer todo tenemos:
import * as nombreDelModulo from ‘./nombreArchivo’

# Proyecto: Buscador de canciones

Si queremos exportar varias cosas hacemos export const nombre1, nombre2, nombre3;
Separando por comas como las variables.

## Patrones de diseño

Soluciones de código reutilizables en diferentes problemas.
Tipos:

- De creación de objetos
- De estructura
- De comportamiento
- De arquitectura

### Module

El más común de los patrones capaz. Es una forma de crear variables públicas y privadas en los objetos.

```
const comprarBoleto = (function(){

    //privado
    let evento = 'Esta es una variable privada'



    //publico
    return {
        mostrarBoleto: function(){
            console.log(evento);
        }
    }
})();
```

### Factory

Es para la creación de objetos. Ayuda a crear objetos qué son similares pero no sabes bien cómo son o cual vas a usar.

```
function ConstructorSitios(){
    this.crearElemento = function(texto, tipo){
        let html;

        if(tipo==='input'){
            html = new InputHTML(texto);

        }else if(tipo === 'img'){
```

```
            html = new ImagenHTML(texto);
        } else if(tipo ==='p'){
            html = new ParrafoHTML(texto);
        }
        html.tipo = tipo;
        return html;
    }
}

const ParrafoHTML = function(texto){
    this.texto = texto
}

const sitioweb = new ConstructorSitios();
sitioweb.crearElemento('textooooooasdlkasdlkasd', 'p');
```

**Singleton**

Es la creación de un objeto qué va a tener UNA SOLA instancia.

```
const alumnos = {
    listaAlumnos:[],

    get: function(id){
        console.log(id);
    },

    crear:function(datos){
        console.log(datos);
        this.listaAlumnos.push(datos)
    }

}

const infoAlumno = {
'nombre':'facha',
'apellido':'diez'
}

const infoAlumno2 = {
    'nombre':'facha2',
```

```
    'apellido':'diez2'
}

alumnos.crear(infoAlumno);
alumnos.crear(infoAlumno2);
```

En lo anterior siempre la instancia de alumnos va a ser una.

**Builder**
Es para crear una abstracción para crear distintos tipos de objetos. Un buen ejemplo es un formulario.

```
import { formularioBuscar } from "./js/interfaz";

class Formulario{
    constructor(){
        this.campos = []
    }

    agregarCampo(tipo,texto){
        let campos = this.campos;

        let campo;
        switch(tipo){
            case "text":
                campo = new InputText(texto);
                break;
            case "email":
                campo = new InputEmail(texto);
                break;
            case "button":
                campo = new Boton(texto);
                break;
        }

        campos.push(campo)
    }
}


class Inputs{
```

```
    constructor(texto){
        this.texto = texto
    }
}

class InputText extends Inputs{
    constructor(texto){
        super(texto)
    }
    crearElemento(){
        const inputText = document.createElement('input');
        inputText.setAttribute('type', 'text');
        inputText.setAttribute('placeholder', this.texto);
        return inputText;
    }
}


formularioBuscar.agregarCampo('text','Agregar nombre');
```

Un ejemplo puede ser hacer una crear una rutina de entrenamientos y con el Builder podemos manejar cuando es una rutina de pesas, de cardio, de hiit, etc.

**Observer**

También se lo conoce como subscriptor-publicador

```
// creamos el observador con los metodos que requerimos
let observer = {
    // suscripcion al publicador
    obtenerOfertas: function(callback){
        if(typeof callback === "function"){
            this.suscribers[this.suscribers.length] = callback;

        }
    },
    cancelarOfertas: function(callback){
        for( let i = 0; i < this.suscribers.length ; i++){
            if(this.suscribers[i] === callback){
                delete this.suscribers[i];
```

```
            }
        }
    },
    publicarOferta: function(oferta){
        for( let i = 0; i < this.suscribers.length ; i++){
            if(typeof this.suscribers[i] === 'function'){
                this.suscribers[i](oferta);
            }
        }
    },
    // creacion de nuevos publicadores
    crear: function(objeto){
        for(let i in this){
            if(this.hasOwnProperty(i)){
                objeto[i] = this[i];
                objeto.suscribers = [];
            }
        }
    }
}

// vendedores o publicadores
const udemy = {
    nuevoCurso: function(){
        const curso = 'Un nuevo curso de Javascript';
        this.publicarOferta(curso);
    }
}

const facebook = {
    nuevoAnuncio: function(){
        const oferta = 'Compra un celular';
        this.publicarOferta(oferta);
    }
}

// crear publicadores
observer.crear(udemy);
observer.crear(facebook);
```

```
const danilo = {
    compartir: function(oferta){
        console.log('excelente oferta!  ' + oferta);
    }
};

const jose = {
    interesa: function(oferta){
        console.log('me interesa la oferta de ' + oferta);
    }
};

udemy.obtenerOfertas(danilo.compartir);
udemy.obtenerOfertas(jose.interesa);
udemy.nuevoCurso();
udemy.cancelarOfertas(jose.interesa);
udemy.nuevoCurso();

facebook.obtenerOfertas( jose.interesa);
facebook.obtenerOfertas( danilo.compartir);
facebook.nuevoAnuncio()
```

**Mediator**

Intermediario para comunicarse con objetos

```
const Vendedor = function(nombre){
    this.nombre = nombre;

}

const Comprador = function(nombre){
    this.nombre = nombre;
    this.sala = null;

}
Vendedor.prototype = {
    oferta: function(articulo, precio){
        console.log(`Tenemos el siguiente articulo ${articulo}, iniciamos en
$${precio}`);
    },
```

```
    vendido: function(comprador){
        console.log(`Vendido a ${comprador} (sonido de mazo)`);
    }
}

Comprador.prototype = {
    oferta: function(mensaje, comprador){
        console.log(`${comprador.nombre}: ${mensaje}`)
    }
}

// subaste es el mediador
// es la que controla los objetos
// vendedores y compadores
const Subasta = function(){
    let compradores = {};
    // las salas es donde van a interactuar los objetos
    return{
        registrar: function(usuario){
            compradores[usuario.nombre] = usuario ;
            usuario.sala = this;
            // console.log(compradores);
        }
    }
}

// instanciar las clases
const juan = new Comprador('Juan');
const pablo = new Comprador('Pablo');
const karen = new Comprador('Karen');

const vendedor = new Vendedor('Vendedor');

const subasta = new Subasta();
subasta.registrar(juan);
subasta.registrar(pablo);
subasta.registrar(karen);

vendedor.oferta('mustang 1996', 3000);
juan.oferta(3500, juan);
```

```
pablo.oferta(4000, pablo);
karen.oferta(10000, karen);
vendedor.vendido(karen.nombre);
```

**Higher Order Functions**
Hacen el código más corto y fácil de leer

**forEach()**

```
arreglo.forEach((elemento, index) =>{
    console.log(`El elemento: ${elemenot} esta en la posicion: ${index}`)

})
```

**Map()**
La diferencia principal es qué map() te retorna un arreglo nuevo con lo que esté adentro.

```
arreglo.map((elemento)=>{
    console.log(elemento)
})
```

**Filter()**
Crea un arreglo basado en una prueba.

```
arreglo.filter(elemento =>{
    return (elemento ===true)
})
```

Va a retornar solo los elementos qué cumplan la condición del return, creando así un arreglo.

**Find()**
El find va a retornar el primer elemento qué cumpla la condición, y solo ese.

```
arreglo.find(elemento => elemento === true)
```

**Reduce()**
Toma todos los valores y retorna la cantidad.

```
arreglo.reduce((total,elemento) => total += elemento.precio, valorInicializar)
//el valor a inicializar generalmente es 0, es inicializar total básicamente
```

**Some()**
Va a evaluar una condición y va a retornar un booleano(true o false). Es un exists.

```
arreglo.some(elemento=> elemento.nombre==='Danilo')
```

# Proyecto: Buscador de autos

Una buena práctica para obtener algo de un input es hacerlo con:
e.target.value

Se pueden agregar cuantos .filter() necesite
Ejemplo:

```
const resultado = obtenerAutos().filter(filtrarMarca).filter(filtrarYear)
```

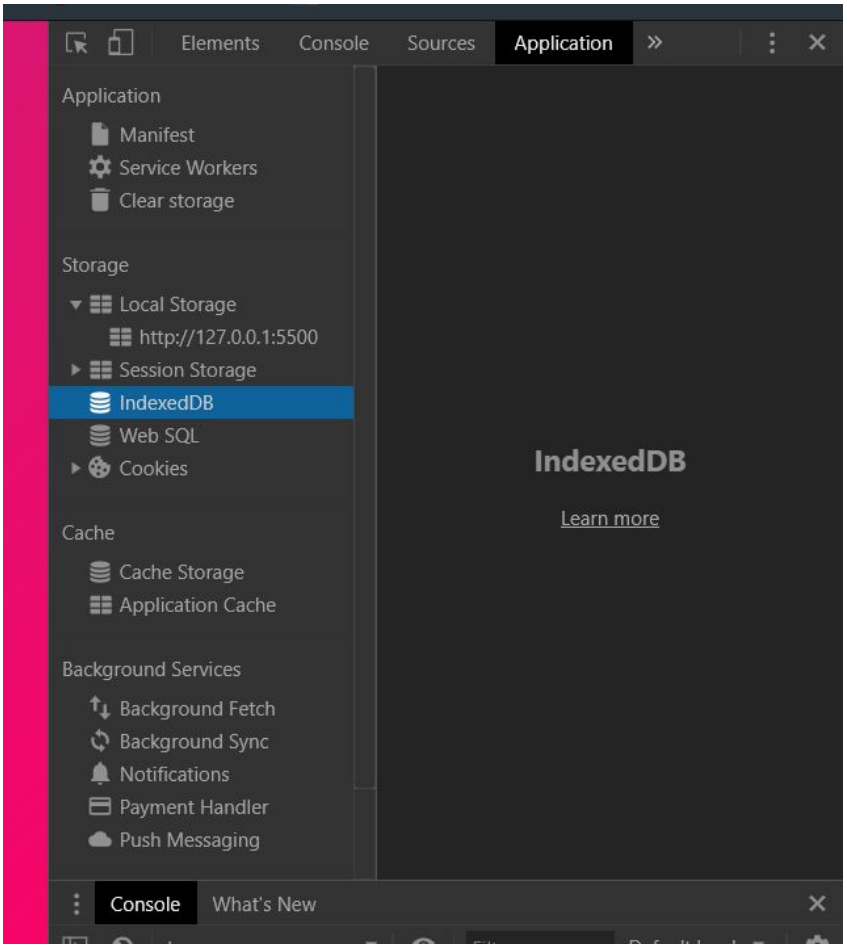
# Proyecto: INDEXED DB, Almacenamiento del lado del cliente

**Storage en el cliente**
Es guardar archivos en el navegador con JS, no guardar nada muy importante, NUNCA guardar información confidencial o sensible.
Tenemos:

- Web storage(session y local storage): Solo pueden guardar strings
- Indexed DB, Firebase, etc: Pueden guardar lo qué sea

**Indexed DB**
Usa indices para almacenar los datos y hacer más rápida la lectura.
Almacena datos estructuras, mucha capacidad.
Es una base de datos completa, guarda lo qué sea. Es asincrono.
NO se puede sincronizar con el backend ni usar con el navegador en privado. Si el usuario limpia el caché y demás, los datos se pierden.

Para trabajar con indexed DB vamos a la consola, y seleccionamos aplicattion:



Todo el codigo va a estar dentro del DOMContentLoaded.

Para crear la base de datos.

```
   document.addEventListener('DOMContentLoaded', () =>{
    //crear la base de datos
    let crearDB = window.indexedDB.open('nombreDeLaBD', numeroDeVersion)


    //si hay un error mandarlo a la consola
    crearDB.onerror = function(){
        console.log('Hubo un error');
    }

    //si todo esta bien, alerta y asignar la D

    crearDB.onsuccess = function(){
       console.log('todo piola')

       DB = crearDB.result;
    }

})
```

El número de DB o versión tiene qué ser un número entero si o si.

Vamos a guardar la BD en la variable DB = crearDB.result

**Creando el esquema de la BD**

```
    //Este metodo solo se ejecuta una vez
    crearDB.onupgradeneeded = function(e) {

    }
```

el onupgradedneeded Se va a ejecutar una sola vez, para no crear una y otra vez el esquema

```
    //Este metodo solo se ejecuta una vez
    crearDB.onupgradeneeded = function(e) {
        //El evento es la base de datos
        let db = e.target.result
```

```
        //definir el objetc store, toma 2 parametros, el nombre de la db y
segundo
        //las opciones

        let objectStore = db.createObjectStore('citas', {keyPath: 'key',
autoIncrement:true})

    }
```

Para crear los índices

```
objectStore.createIndex('mascota', 'mascota', {unique: false})
```

Los parámetros son, el nombre, la llave, y las opciones.

Tenemos qué mapear las inserciones a un objeto. Después podemos hacer la inserción en la BD.
Para empezar a hacer las inserciones tenemos las transacciones

```
        let transaction = DB.transaction(['citas'],'readwrite');
        let objectStore = transaction.objectStore('citas');
```

```
        let peticion = objectStore.add(nuevaCita);



        peticion.onsuccess = () => {
            form.reset();
        }
        transaction.oncomplete = () => {
            console.log('cita agregada')
        }
        transaction.onerror = () => {
            console.log('Hubo un error')
        }
```

Una buena forma de agregar funciones al html es

```
botonBorrar.onclick = borrarCita
```

**Novedades de ES10**

**.flat()**
Funciona para un arreglos de arreglos en una lista de números.
Le damos como parámetro cuantos niveles queremos analizar, en una arreglo de arreglo, ejemplo
[1, 2, 3, [4 , 5]]
Haríamos:

```
arreglo.flat(1)
```

Y eso devolveria
[1, 2, 3, 4, 5]
Cambiamos el 1 por la cantidad de niveles qué tengamos.
Si tenemos muchos niveles y no queremos perder tiempo en eso le damos el parametro Infinity

**.flatMap()**
Sirve para mapear un elemento y retornar un arreglo. Es la combinación de ambos.

**.fromEntries()**
Permite crear un objeto desde un set o un mapa.

**.trimStart() y .trimEnd()**
Te permiten eliminar el espacio en blanco de un string.

```
const correo = '           @gmail.com'
correo.trimStart()

const correo = '@gmail.com              '
correo.trimEnd();

const correo = '        @gmail.com             '
correo.trim();
```

**.toString()**
Retorna cualquier cosa con qué lo llames en un string, incluso una función. Es útil para debbuguear o para hacerlo con una función qué descargas de una librería o algo así.

**try y catch**
El try catch es para administrar los errores.

```
try {
    throw new Error('Algo salio mal, F');
} catch (error) {
    console.log(error);
}
```

# JavaScript Avanzado

### .this con Implicit Binding

```
const usuario = {
    nombre:'Danilo',
    edad: 20,
    presentacion(){
        console.log(`Mi nombre es ${this.nombre}`)
    }

}
```

### .this con Explicit Binding

Usamos el call para darle contexto a la función, es decir darle un objeto con el qué usar el this. Cuando usamos call con un arreglo hay qué darle si o si las posiciones

```
function presentacion(el1, el2){
    console.log(`Mi nombre es ${this.nombre} y me gusta el ${el1} y el
${el2}`)
}

const usuario = {
    nombre:'Danilo',
    edad: 20,

}

const musica = ['trap', 'pop']

presentacion.call(usuario, musica[0], musica[1]);
```

También podemos usar .apply

```
presentacion.apply(usuario, musica)
```

Y tenemos .bind qué es igual a .call pero hay que asignarlo a una nueva función si o si.

```
const nuevaFn = presentacion.call(usuario, musica[0], musica[1]);

nuevaFn();
```

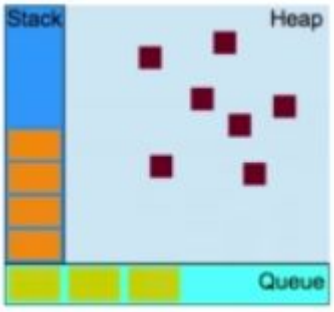
### .this con Window Binding

Cuando una función no tiene implícitamente el this va a buscar en algo llamado window el valor qué requiera:

```
function obtenerAuto(){
```

```
    console.log(`Mi auto es color: ${this.color}`)
};

const color = 'negro'; //nofunciona
window.color = 'negro'; //se llama correctamente
```

**Event Loop**




Se ejecuta primero los eventos del stack y después los del queue.

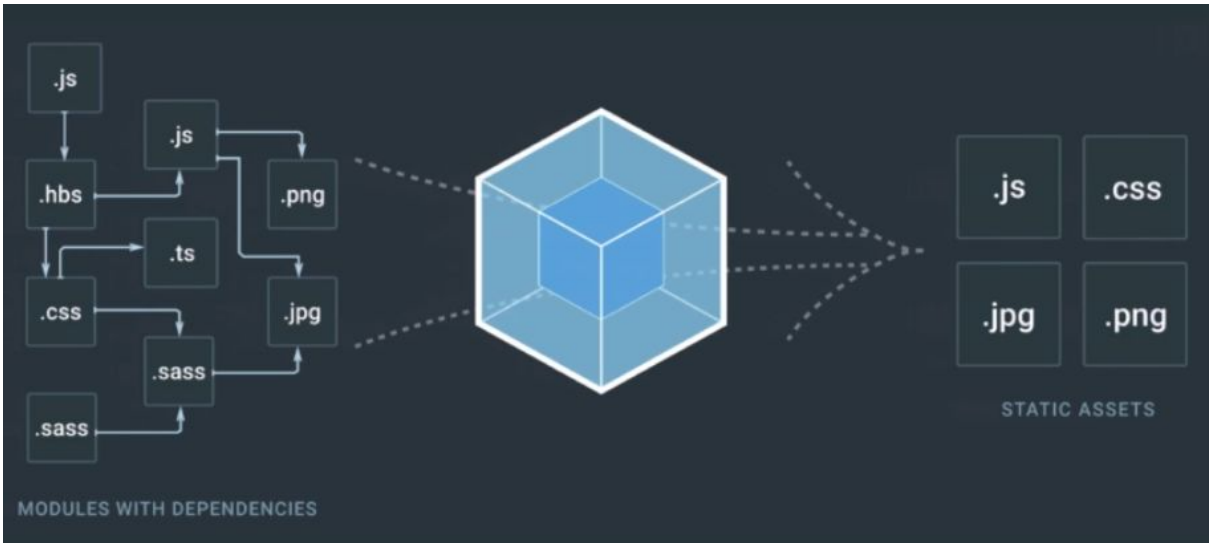
El setTimeOut coloca a las instrucciones dentro del queue. Así qué siempre se va a ejecutar primero lo del stack y después lo del setTimeOut.
Los promises se ponen antes de las cosas del Queue, porque están antes del task queue.

Prioridad en orden:

- stack
- promises en job queue
- cosas en el task queue

# Webpack

Es un paquete de módulos para aplicaciones moviles. Procesa la aplicación y mapea todas las dependencias de un módulo creando uno o varios paquetes nuevos.




Existen 4 conceptos claves:

- Entry: punto de entrada qué especifica qué módulo tiene qué usar webpack para crear la gráfica de dependencias. /src
- Output: en qué parte se almacena el paquete qué se creo. /dist
- Loaders: con estos se pueden cargar varios formatos como sass, less, imagenes o html.
    - test: qué archivos se van a transformar
    - use: qué loader usar para un archivo determinado
- Plugins: realizan tareas como optimizar el paquete, administrar los assets, entre otros…

Para trabajar con webpack necesitamos NPM y NodeJS.

**Creando un archivo de dependencias json**
Entramos a la consola:
Escribimos npm init y damos enter a todo.

Para instalar dependencias es:
npm install nombreDependencia --save-dev
npm install webpack --save-dev en esté caso

Para instalar la consola de webpack:
npm install webpack-cli --save-dev

Generalmente el npm install va a instalar todos los módulos de node, pero si tenemos en el json especificadas cuales son y ponemos npm install solo va a instalar las qué están ahí clave

**Creando un bundle**
Tenemos qué crear una carpeta **src** para crear adentro los .js .sass, etc.
Creamos un index.js y hacemos
node_modules\.bin\webpack\ src\index.js en consola

Nos va a crear un main.js con lo siguiente:


Eso anterior es nuestro primer bundle

**Creando un archivo de configuración**
Para eso creamos dentro del src un archivo llamado:
webpack.config.js
En el config escribimos esto

```
const path = require('path')  //variable de Node

module.exports = {
    entry: '.\src\index.js', //entrada que es los archivos a convertir en
bundle
    output: {   //salida
        filename: 'bundle.js',  //como se va a llamar el archivo
        path: path.join(__dirname, '\dist')},    //donde se va a guardar
                                                 //__dirname es la carpeta
en la que estemos
                                                 //parados

}
```

Y ejecutamos por consola
node_modules\.bin\webpack

**Crear un script para ejecutar Webpack**
Vamos al package.json en la parte de “**Scripts**”,

```
  "scripts": {
    "webpack --mode development"
}
```

y luego en consola

npm run ejecutar

**Transpilar código con Babel**
Babel permite “compilar”, el creador de C está muy enojado, nuestro código de JS para qué incluso navegadores viejos lo puedan leer, también es muy útil cuando estamos usando alguna funcionalidad muy nueva de JS y no sabemos si va a correr en todos los navegadores.

Para instalar hacemos por consola

npm install --save-dev @babel/cli @babel/core @babel/preset-env @babel/register babel-loader

Después en la carpeta src creamos un archivo .babelrc y escribimos

```
{
    "presets":["@babel/preset-env"]

}
```

Después tenemos qué agregar Babel a los modulos del config. Se hace abajo del output.

```
    module:{
        rules: [
            {
                test: /\.js$/,
                exclude: /node_modules/,
                use:{
                    loader: 'babel-loader'
                }
            }

        ]

    }
```

Agregamos el script watch a package.json

```
"watch": "webpack --w --mode development "
```

y ponemos en consola npm run watch
Lo qué hace watch es escuchar por los cambios qué se hagan en el código y se recargue el navegador. Para cancelar watch hacemos Ctrl+C

**Importar css en JS con Webpack**
Instalamos los paquetes correspondientes
npm install --save-dev style-loader css-loader

Ahora agregamos las reglas:

```
            {
                test: /\.css$/,
                exclude: /node_modules/,
                use:[
                {loader: 'style-loader'},
                {loader: 'css-loader'},
                ]
            },
```

Y ahora tenemos qué agregar en nuestro archivo .js lo siguiente

```
import '../carpetaCSS/style.css';
```

Y listo cuando demos npm run ejecutar se va a exportar el paquete con css.

**Creando múltiples bundles**
Ahora nuestro entry va a ser un objeto

```
    entry: {

    index:'.\src\index.js',
    nosotros:'.\src\nosotros.js'

}
```

le damos el nombre de cada módulo

Después para qué sea por ejemplo index.bundle o nosotros.bundle hacemos

```
filename: '[name].bundle.js'
```

**Agregando common chunks**
Para evitar todo el codigo qué se repita del bundle grande. Se usa un plugin llamada

common chunks. Ponemos lo siguiente después de los outputs. El test es los archivos qué va a buscar, name es como se va a llamar y chunks es para qué comprima todo.

```
    optimization: {
        splitChunks:{
            cacheGroups:{
                commons: {
                    test: /[\\/]node_modules[\\/]/,
                    name: 'common',
                    chunks: 'all'

                }
            }
        }
    }
```

NOTA: Reduce zarpado la cantidad de código generado.
A ese common hay qué llamarlo desde el html.

**Webpack dev-server**
npm install --save-dev webpack-dev-server
Es como un live server pero más copado.
Agregamos al config en el mismo nivel de output,entry, etc…

```
    devServer:{
        contentBase:path.join(__dirname,'dist'),          //carpeta de donde
se sacan los archivos
        compress: true,
        port: 9000,


    },
```

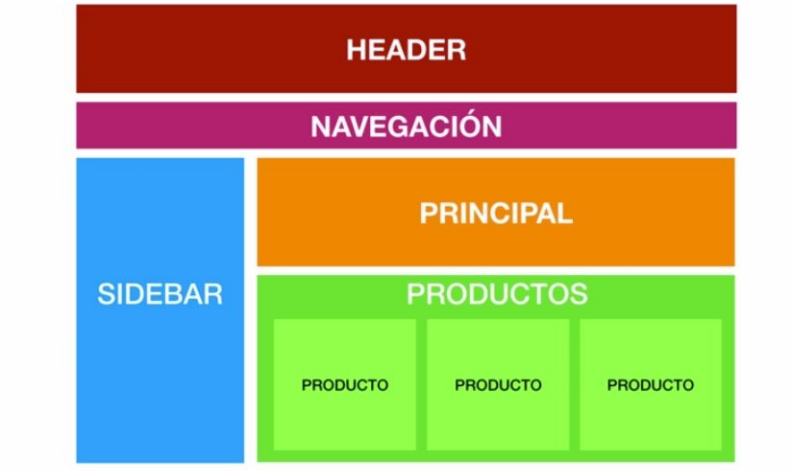
Ahora vamos al package.json

```
"dev": "webpack-dev-server --mode development --open"
```

Después npm run dev y ahí va a abrir un live server
**ES MUY IMPORTANTE TENER TODOS LOS ARCHIVOS EN DIST**

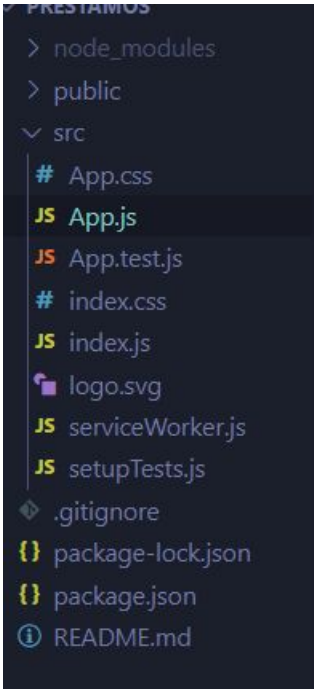
# REACT!!!!!!!!!

Es una librería para crear interfaces de usuario para apps web. Corre en el cliente, no es necesaria una respuesta del servidor.
Usa **componentes** qué son modulos de codigo reutilizable, en lo siguiente cada cosa va a ser un componente.




Tenemos qué crear una app de react.
npx create-react-app my-app
cd my-app
npm start

Se escribe el código generalmente en src, así luce una carpeta recién creada:



Agregamos skeleton

```
 <link rel="stylesheet"
href="https://cdnjs.cloudflare.com/ajax/libs/skeleton/2.0.4/skeleton.min.css"
integrity="sha256-2YQRJMXD7pIAPHiXr0s+vlRWA7GYJEK0ARns7k2sbHY="
crossorigin="anonymous" />
```

en index.html de Public

Lo de public es lo qué se muestra.
Y la dependencia normalize

```
    <link rel="stylesheet"
href="https://cdnjs.cloudflare.com/ajax/libs/normalize/8.0.1/normalize.min.css
" integrity="sha256-l85OmPOjvil/SOvVt3HnSSjzF1TUMyT9eV0c2BzEGzU="
crossorigin="anonymous" />
    <title>React App</title>
```

Index.js solo importa básicamente lo qué tengamos en App.js qué es lo principal

**COMPONENTES**
Por organización está bueno crear una carpeta de componentes.
Los componentes tienen qué iniciar con mayúscula y ser un archivo JS.

Siempre arrancamos importando react. Tenemos el atajo **imr**

```
import React from 'react';
```

Un componente siempre es una función, y en el return ponemos lo qué se va a mostrar. Es importante también exportar la función. Entonces la estructura básica queda así:

```
import React from 'react';

function Header(){

    return(
        <h1>Hola mundo</h1>

    )
}

export default Header;
```

Ahora en el archivo App tenemos qué llamar nuestros componentes. Primero lo importamos, y después para escribirlo tenemos qué hacerlo en XML.

```jsx
import React from 'react';
import Header from './componentes/Header'; //acá importamos

function App() {
  return (
    <div className="App">
      <Header /> //Este es el componente creado arriba
    </div>
  );
}

export default App;
```

Es muy importante saber qué no podemos retornar varios elementos. Siempre hay qué retornar UNO qué los contenga a todos en el caso de hacer varios.

Algo qué podemos hacer en la app para no poner todo dentro de un div, es crear un Fragment, un fragment se va a comportar como un div pero no va a aparecer en el frontEnd.

```jsx
import React, {Fragment} from 'react';
import Header from './componentes/Header';

function App() {
  return (
    <Fragment>
      <Header />
    </Fragment>
  );
}

export default App;
```

**Pasar datos entre componentes**
Los datos fluyen de los componentes principales a los hijos. Se hace por medio de algo llamado **props**.
React usa JSX qué permite mezclar Javascript con HTML
Todo lo qué pongamos entre llaves se va a interpretar como código JS

Ahora usamos props para darle titulo a nuestro H1, en la APP:

```
      <Header
        titulo= "Prestamos"


      />
```

En el Header:

```
    return(

        <h1>{props.titulo}</h1>
    )
```

**Otra forma para escribir Componentes**
Simple function snippets, en el vscode escribimos **sfc** y nos crea la estructura para escribir componentes

```
const Componente2 = (props) => {
    return ( <h1>Hola</h1> );
}
```

**Creando un formulario**
Hay qué tener en cuenta qué React no nos deja usar class para las clases de HTML, tenemos qué usar className

Lo qué hace qué React sea tan rápido es el uso de State.

```
import React, {useState} from 'react'
```

Lo importamos

Después definimos las variables para usarlos

```
    const [cantidad, guardarCantidad] = useState(0);
```

- En cantidad va el nombre del dato o la variable qué vamos a guardar
- En guardarCantidad, va la función qué usemos para obtener los datos.
- El 0 es para el valor inicial, podría ser un string vacío.

JSX tiene muchos eventos, ahora usamos onChange para qué escuche por lo qué escribamos en el formulario.

```
                    onChange={leerCantidad}
```

Dentro va una función.

```
    const leerCantidad = ()=>{
        return console.log('Escribiendo cantidad...')
    }
```

Acá tenemos la lectura con la función guardarCantidad qué asignamos en el State

```
   const [cantidad, guardarCantidad] = useState(0);
   const leerCantidad = (e)=>{
       guardarCantidad(parseInt(e.target.value))
   }
```

Una mejor manera de hacerlo para no hacer una función leerCantidad extra es meter todo dentro del onChange

```
                    onChange= {e => {
                      guardarCantidad(parseInt(e.target.value))
                  }}
```

Los valores NUNCA pasan del hijo al padre, pasan del padre al hije
Entonces vamos a definir el estado en la App.js así:

```
function App() {

    //definir el State
    const [cantidad, guardarCantidad] = useState(0);
```

Y tenemos que pasar mediante props desde la App al Formulario

```
    <Formulario
    cantidad = {cantidad}
    guardarCantidad = {guardarCantidad}

    />
```

**Validar el formulario**
El evento de React para submit es **onSubmit**

```
<form onSubmit={calcularPrestamo}>
```

**Operadores ternarios re locos**

```
{(error ? <p className="error">Todos los campos son
```

```
obligatorios</p>:'')}
```

si error === true entonces muestra el párrafo sino no

Para agregar lógica a React conviene tener por un lado los Componentes y por otro lado los **Helpers** qué son módulos de js con funciones.

**Carga condicional de componentes**

```
    let compoenente;
    if(total === 0){

      componente = <Mensaje/>
    }else{
      componente = <Resultado/>
    }
```

```
    <div className="mensajes">
    {componente}
    </div>
```

**Deployment - Subirlo a un server**
Usar netlify

Hacemos en consola npm run build

# Introducción a NODEJS

Para crear un servidor con express hacemos index.js en la carpeta server:

```
//Importar express
//Esta es una version vieja de importar y exportar MODULOS
const express = require('express');




const app = express();
const port = 3000;
//Configurar express

app.use('/', (req, res) => {

   res.send('Hola mundo en NODEJs')



});

app.listen(port)
```

Y agregamos un script al package.json

```
"start": "nodemon server"
```

**IMPORTANTE: NODE MANDA EL CONSOLE.LOG A LA CONSOLA DE LA PC NOOOOO AL NAVEGADOR.**

**REQUEST REQ: ES LO QUÉ PEDIMOS A LA PÁGINA**
**RESPONSE RES: ES LO QUÉ MANDAMOS A LA PÁGINA**

Para crear las rutas de las diferentes paginas hacem

```
app.use('/', (req, res) => {
app.use('/nosotros', (req, res) => {
app.use('/tuviejaETC', (req, res) => {
```

**TEMPLATE ENGINES - PUG - HAMJS**

**EN LA CARPETA SERVER/VIEWS TENEMOS TODO EL MANEJO DE LAS VISTAS CON**

**HTML CSS Y JS**

**DENTRO DE VIEWS HAY QUÉ CREAR UNA CARPETA POR CADA PAGINA**

**Y TAMBIÉN DENTRO DE VIEWS CREAMOS UNA CARPETA LAYOUT CON UN INDEX QUÉ NOS VA A SERVIR PARA LA MASTER PAGE**

**Para convertir html a pug es muy HTML2JADE.ORG**

**CLAVE SEQUELIZE PARA MANEJAR BDS EN NODE**
**https://sequelize.org/**