1. Introducción a JavaScript

Anotación: Hay ejemplos prácticos en los videos que no realicé al ser temas que ya sé y no me hacen falta repasar. Por eso algunas carpetas pueden estar vacias.

Vincular JS con HTML: https://www.w3schools.com/tags/att_script_src.asp

1. Sintaxis de JavaScript

Es case senstitive

Es de tipado debil/dinamico: Las variables son del tipo de dato que almacenan.

Las sentencias finalizan con ;. No es obligatorio pero es muy recomendable.

Los bloques finalizan con la lave de cierre '}'. De forma opcional se puyede añadir un ; despues de la llave.

2. Variables y constantes scope o ámbito

Una variable es un espacio que reservamos en memoria para almacenar un dato que podrá cambiar durante la ejecución de nuestro programa.

La palabra reservada es "let", no es recomendable usar "var".

Las variables se pueden: declarar, inicializar y modificar.

Una constante es un espacio que reservamos en memoria para almacenar un dato que no cambiará durante la ejecución de nuestro programa.

La palabra reservada para declarar constantes es "const".

El scope o ámbito es la zona donde existe nuestra variable o constante.

3. Declaración, inicialización y Modificación

Una variable se declara con la siguiente estructura:

- let nombre;

Una variable se inicializa con la siguiente estructura:

- numero = 5;

Se puede declarar e inicializar en la misma sentencia:

- let numero = 5;

Para modificar el valro de una variable existente:

- numero = 3;

Las constantes solo admiten la declaración e inicialización en la misma sentencia. Se modifica unicamente en su declaración.

- const PI = 3.14;

4. Tipos de datos principales en JavaScript

- Primitivos.
  - Numeros -> let numero = 5;
  - Strings (cadenas) -> let palabra = "hola"; let palabra = 'hola';
  - Boolean -> let respuesta = true; let respuesta = false;
  - Undefined.
  - Null.
  - Symbol.

5. Tipos de operadores

- Matématicos
  - Suma/Concatenación: +.
  - Resta: -.
  - Multiplicación: *.
  - División: /.
  - Módulo: %.
  - Asignación
  - Asignación: =.
  - Suma y asignación: +=.
  - Resta y asignación: -=.

- Multiplicación y asignación: *=.
    - División y asignación: /=.
    - Módulo y asignación: %=.
- Incremento/Decremento
    - Post-incremento en 1: x++.
    - Pre-incremento en 1: ++x.
    - Post-decremento en 1: x--.
    - Pre-decremento en 1: --x.
- Operadores Lógicos
    - Igualdad: ==.
    - Desigualdad: !=.
    - Estrictamente iguales: ===. (Mismo valor y mismo tipo).
    - Estrictamente desiguales: !==. (Mismo valor y mismo tipo).
    - Mayor que: >.
    - Menor que: <.
    - Mayor o igual que: >.
    - Menor o igual que: <.

6. Strings

Método: Es todo aquello que la cadena puede hacer. Ej: Convertirse en mayúsculas.

Propiedad: Son caracteristicas que al cadena tiene por ser una cadena. Ej: Longitud.

Se cumplen utilizando la siguiente nomenclatura, al igual que con todos los objetos:

- string.método().
- string.propiedad.

Documentación de métodos y propiedades: https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/String

Los template string se engloban dentro de los caracteres ``, dentro funciona de igual manera que en C#.

7. Objeto Math

Documentación de Math: https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Math

El objeto Math se utiliza para hacer operaciones matemáticas complejas/específicas.

Al ser un objeto también utiliza la nomenclatura del punto.

8. Condicionales

El flujo de un programa siempre será de arriba hacia abajo.

Estructurasn de control de flujo:

- Condicionales
  - Simples
  - Compuestos
  - Múltiples
- Bucles
  - Determinados
  - Indeterminados

La estructura switch tiene una sintaxis múltiple, es decir, puedo tener muchos case sin break, se van a ejecutar todos hasta encontrar un break o terminar con el programa.

Ejemplo encadenado:

```
let num = 1;
switch(num){
  case 1:
    num++;
  case 2:
    num++;
```

```
    case 3:
        num++;
    case 4:
        num++;
    case 5:
        num++;
}
console.log(num); -> Será 6.
```

Ejemplo sin encadenar

```
let num = 1;
switch(num){
    case 1:
    case 3:
    case 5:
        console.log("es impar");
        break;
    case 2:
    case 4:
        console.log("es par");
        break;
}
```

9. Operador ternario

Se utiliza cuando una condición será true o false, al igual que el if.

Su ejecución puede tener una ovarias sentencias, en este caso irán separadas por coma y entre paréntesis. Puede tener sentencias compuestas

```
Condición ? (true) : (false)
Condición ? ((true, true) : (false, false))
```

Ejemplo: (2 % 2 === 0) ? console.log("Es par") : console.log("Es impar")

10. Arrays

Documentación de Array: https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Array

Son estructuras que nos permiten almacenar varios datos y agruparlos.

Se pueden llenar con cualquier tipo de dato válido en JavaScript y deben ir separados por comas.

Se pueden mezclar tipos de datos en un mismo array, pero no es recomendable.

Se declaran con llaves cuadradas o corchetes.

Pueden declararse vacios o con contenido ya establecido.

Pueden añadirse o eliminarse elementos en el momento que queramos (Funcionan como listas).

Cada uno de los elementos podrá ser identificado por su índice, es decir por su posición.

Los índices empiezan a contar desde el 0.

Ejemplos de declaración:

- let numeros = [];
- let numeros = [x, z, y, b, ...];
- let numeros = new Array();

11. Bucles

Se usan cuando queremos que un trozo de código se repita.

Existen bucles determinados e indeterminados.

Los determinados se usan cuando especificamos el número de veces que se va a repetir.

- Ej: Imprimir números del 1 al 10.

Los indeterminados los utilizamos cuando no sabemos el número de veces que se va a repetir.

- Ej: Repetir mensaje de introducir contraseña.

La estructura de un bucle es siempre la misma.

- Ej: Bucle { Código a ejecutar }.

En EMASCRIPT 6 llegaron los bucles for in y for of.

- La palabra of devolverá el valor del elemento en el array.
- La palabra in devolverá el index del elemento en el array. Podemos sacar el contenido como por ejemplo: array[variable_local]; (Como en C).

2. Objetos, funciones y métodos

1. Objetos introducción

Son estructuras de datos que representan propiedades, valores y acciones que puede realizar el objeto.

Todos los objetos tiene propiedades o atributos y comportamientoes o acciones representados por pares key:value.

Para acceder a las propiedades y acciones del objeto se utiliza la nomeclatura del punto. Otra forma es: objeto['atributo'].

2. Funciones introducción

Son fragmentos de código que escribimos para ejectuar una tarea y no volver a escribir el mismo código más de una vez.

Nos ayuda a modularizar el código.

Las funciones deben realizar una sola tarea.

La sintaxis puede ser:

- function NOMBRE () {CODIGO A EJECUTAR};
- const NOMBRE = () => {CODIGO A EJECUTAR};

Pueden recibir parámetros.

Pueden devolver valores.

Con la palabra return devolvemos un valor.

En la funcion const al utilizar lamba existe un return implicito si hay una única instrucción, también se pueden obviar las llaves. En caso de tener mas de una linea se utiliza return.

Ejemplo: const NOMBRE = (p1, p2) => p1 + p2; -> Return implicito

3. Programación Orienteada a Objetos - Clases

Es un paradigma de la programación. Un paradigma es una forma de resolver un problema.

Algunos de los conceptos fundamentales son:

- Clase
- Herencia
- Objeto
- Método
- Evento
- Etc

Las clases necesitan una función constructora. Se tiene que llamar al constructor y se ejecuta cuando creemos un objeto (Desconozco si hay una clase por defecto que inicializa todo en 0 en caso de no haber constructor como en .NET).

La sintaxis es:

```
class NOMBRE_DE_LA_CASE{
    constructor(PARAMETROS){
        this.ATRIBUTO
    }
}
```

Los objetos pueden tener funciones asociadas a él. En ese caso se lo denomina 'Métodos'.

Para crear un objeto utilizando la clase se hace con la palabra reservada new y el nombre de la clase que queremos utilizar.

Una vez INSTANCIADO el objeto podremos acceder a sus propiedades y métodos utilizando la nomeclatura del punto o buscando su propiedad en el objeto.

Existen los getter y setters para acceder a los atributos a través de las propiedades get y set.

4. Spread Operator

El spread operator u operador de expansión principalmente expande el contenido de un array. Se podría decir que devuelve el valor de cada index por separado.

Utiliza ... para al comienzo para indicar que es un Spread Operator.

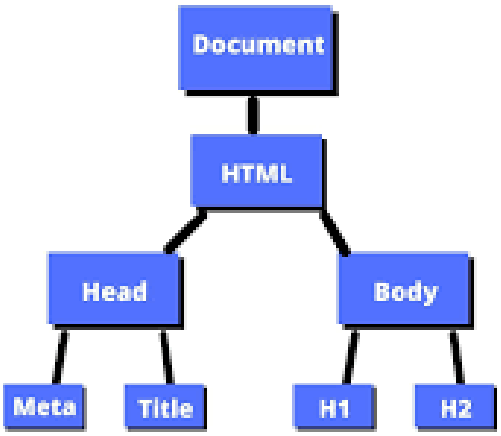
Nos permite que una función pueda recibir una cantidad indefinida de parametros con el concepto Rest parameters: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/rest_parameters

Por ejemplo:

```
const numbers = [12, 2, 3, 23, 43, 2, 3];

console.log(...numbers); | OutPut => 12 2 3 23 43 2 3
```

3. DOM Document Object Model



1. DOM - Introducción

Documentación: https://www.w3schools.com/js/js_htmldom.asp

Se considera DOM toda la estructura HTML de un documento.

No es JavaScript, es una API (Application Programing Interface) que se utiliza a través de JavaScript que viene integrado con el navegador.

El DOM establece una jerarquia a través de nodos, constituyendo algo parecido a un arbol inverso.

Cada parte del arbol del documento es un NODO.

Hay varios tipos de nodos, los más utilizados son:

- Element node - 1 (Cualquier etiqueta HTML).
- Text node - 3 (El contenido de la etiqueta HTML).
- Comment node - 8 (Cualquier comentario en HTML).

Para vincular JavaScript con HTML utilizamos la etiqueta <script src"PATH"></script>. Se recomienda colocarla la linea anterior al cierre del body para que el script pueda acceder a todos los elementos declarados.

2. Selección de elementos

Los elementos se guardan en una constante porque el elemento no van a cambiar.

Los pseudo-elementos no existen en el DOM, sin embargo hay formas de acceder a ellos que veremos más adelante.

document.getElementById("ID") - Acceder a un elemento a través de su id.

document/element.querySelector("SelectorCSS") - Accede al primer elemento que coincida con el selector CSS.

document/element.querySelectorAll("SelectorCSS") - Accede a todos los elementos que coincidan con el selector CSS, devuelve un nodeList.

Para los selectores CSS utilizamos el . al principio para indicar que es una clase.

3. Atributos y clases

Atributos

- element.getAttribute("attribute"); | -> Nos devuelve el valor del atributo del elemento.
- element.setAttribute("attribute", value); -> Nos permite cambiar el valor del atributo del elemento. Tambien podemos setear el valor de un atributo que no haya sido declarado en la etiqueta en un comienzo. Ej: element.setAttribute("placeholder", "Ingrese su nombre");

Clases

- element.classList.add("class", "class", ...); | -> Añadimos una o más clases.
- element.classList.remove("class", "class", ...); | -> Remueve una o más clases.
- element.classList.toggle("class" [,force]); | -> Deshabilita o habilita una clase. Toggle devuelve true si contiene esa clase, en caso de contenerla no añade la clase especificada en el force y viceversa con el false.
- element.classList.contains("class"); | -> Devuelve true o false en función si tiene la clase o no.
- element.classList.replace("oldclase", "newclass"); | -> Sustituye la clase del primer parametro por la del segundo.

Atributos directos más importantes

- id
- value

4. Eventos de ratón y teclado

Documentación: https://developer.mozilla.org/es/docs/Web/Events

Un evento es cualquer cosa que sucede en nuestro documento. Puede ejecutarlo un usuario o la misma programación de la página.

La sintaxis correcta para el mantenimiento y lectura de una página web sería: Element.addEventosListener('event', callback);

El callback es una función anonima que se ejecuta cuando se dispara el evento.

5. Objeto evento

El objeto evento vive siempre y cuando haya un evento.

Ejemplo del objeto evento: input.addEventListener("keyup", (e) => console.log(e));

El objeto evento nos permite saber donde hicimos click. Por ejemplo si tenemos un div dentor de otro div, podemos saber donde se hizo click y a que objeto interno afecto. Ejemplo en el ejercicio 4 del curso-

Los eventos los podemos escuchar (addEventListener) y también los podemos forzar. Para forzarlos simplemente utilziamos la sintaxis element.evento().

Podemos evitar el comportamiento por default de los eventos utilizando .preventDefault(). Esto puede ser muy util para bloquear el envio de un formulario al hacer click en un boton de "Enviar".

6. Crear e instanciar elementos

- Crear un elemento: document.createElement(element)
- Escribir texto en un elemento: element.textContent = texto
- Escribir HTML en un elemento: element.innerHTML = código HTML
- Añadir un elemento al DOM: parent.appendchild(element);
- Añadir fragmentos de código: document.createDocumentFragment();

Ejemplos prácticos y buenas prácticas en el ejercicio correspondiente.

Utilizar un bucle for e insertar HTML en el innerHTML hace que cada vuelta del bucle estamos inyectando codigo HTML en el DOM. El DOM tiene aspecto de arbol de nodos, cada vez que inyectamos de esta manera se esta redibujando ese arbol por completo, lo cual hace que sea ineficiente al gastar muchos recursos del navegador. Para evitar el inconveniente del for, se creo el document.createDocumentFragment(); no sobrecargamos el navegador al hacer este tipo de operaciones ya que simplemente agregamos un nodo en el arbol y no redibujamos el arbol entero por cada bucle del for.

7. DOM - Recorrerlo (DOM Traversing)

El DOM traversing (atravesar el DOM) nos permite situarnos en un nodo y poder movernos entre los mismos.

En caso de no existir un padre, hijo, hermano o cercano devuelve null.

Se pueden combinar cada una de las propiedades y/o repetirlas.

Las propiedades sin la palabra element devuelven tanto los elementos como texto "basura" (en realidad es un salto de linea del documento HTML entre cada tag, lo que hace engorroso leer el elemento). Salvo casos especiales se recomienda usar los que contengan la palabra element.

Cada una de las propiedades debe estar precedida por el nodo al cual hacemos referencia. Ejemplo NODO.propiedad.

Verificar los ejemplos en la carpeta correspondiente para clarificar.

Padre - parent (Nodo del que desciende).

- parentNode: Devuelve el nodo padre (que puede no sr un elemento).
- parentElement: Devuelve el nodo del elemento padre.

NOTA: Los nodos tipo Document y DocumentFragment nunca van a tener un elemento padre, parentNode devolverá siempre null.

Hijos - child (Nodo que desciende de un padre)

- childNodes: Devuelve todos los nodos hijos.
- children: Devuelve todos los nodos elementos hijos.
- firstChild: Devuelve el primer nodo hijo.
- firstElementChild: Devuelve el primer nodo elemento hijo.
- lastChild: Devuelve el último nodo hijo.
- lastElementChild: Devuelve el último nodo elemento hijo.
- hasChildNodes(): Devuelve true si el nodo tiene hijos y false si no tiene.

Hermanos - sibilings (Nodo al mismo nivel)

- nextSibling: Devuelve el siguiente nodo hermano
- nextElementSibling: Devuelve el siguiente nodo elemento hermano.
- previousSibling: Devuelve el anterior nodo hermano
- previousElementSibling: Devuelve el anterior nodo elemento hermano.

Cercano

- closest(selector): Selecciona el nodo más cercano que cumpla con el selector, aún es experimental.

8. Insertar, clonar y borrar elementos

Anteriormente vimos como colocar de manera correcta elementos HTML a través del DOM JS, sin embargo, solo vimos como locarlos al final del elemento al cual le hacemos un append. A continuación veremos como colocar un elemento en cualquier lugar de nuestro HTML, clonar y remover nodos del DOM.

Muchas veces vamos a necesitar acceder a un children del parent para poder colocarlo antes o despues dentro del parent, y que no salga salvo que lo querramos.

Al clonar debemos tener cuidado de no repetir un ID.

Verificar los ejemplos en la carpeta correspondiente para clarificar.

Insertar y eliminar elementos 2

- parent.insertBefore(newElement, referenceElement): Insertar un elemento antes del elemento de referencia.

SOPORTE TOTAL:

- parent.insertAdjacentElement(position, element):
- parent.insertAdjacentHTML(position, HTML):
- parent.insertAdjacentText(position, text):

positions:

- beforebegin: Antes de que empiece (hermano anterior).
- afterbegin: Despues de que empiece (primer hijo).
- beforeend: Antes de que acabe (último hijo).
- afterend: Después de que acabe (hermano siguiente).

- parent.replaceChild(newChild, oldChild): Reemplaza un hijo por otro

DOM manipulation convenience methods - JQuery Like:
https://caniuse.com/#search.jQuery-like

positions:

- parent.before(Element): Antes de que empiece (hermano anterior).
- parent.prepend(Element): Despues de que empiece (primer hijo).
- parent.append(Element): Antes de que acabe (último hijo).
- parent.after(Element): Después de que acabe (hermano siguiente).
-
- child.replaceWith(newChild)

Clonar y eliminar elementos

- element.cloneNode(true|false): Clona el nodo. Si le pasamos true clona todo el elemento con los hijos, si le pasamos false clona solo el elemento sin los hijos.
- element.remove(): Elimina el nodo del DOM.
- element.removeChild(child): Elimina el nodo hijo del DOM.

9. Objetos nativos y timers

Objeto window - Es el objeto global, de él descienden directa o indirectamente todos los objetos.

Algunos ejemplos directos

- No se recomienda utilizarlas ya que son ventanas que pausan la carga/ejecucion de la pagina hasta que se resuelvan. Salvo casos concretos no se recomienda utilizar.
- alert() -> Genera un pop-up de alerta con un boton de aceptar.
- prompt() -> Genera un pop-up con un textbox para ingresar datos con un boton de aceptar y otro de cancelar.
- confirm() -> Genera un pop-up que devuelve true si acepta y false si cancela. No puede ser rediseñado con CSS.

Objeto console - Es el objeto que contiene la consola del navegador. Todos los navegadores tienen su propia consola.

https://developer.mozilla.org/es/docs/Web/API/Console

Algunos ejemplos

- console.log() -> Imprime información en consola.
- console.dir() -> A veces el navegador cambia la manera de mostrar la información y no nos deja desplegar las propiedades de, por ejemplo, un elementO.

  Para ello con el dir podemos obligar a que siempre aparezca de esa forma.
- console.error() -> Imprime mensajes de error.
- console.table() -> Imprime una tabla indice:valores.

Objeto location - Es el objeto que contiene la barra de direcciones.

https://developer.mozilla.org/es/docs/Web/API/Location

Algunos ejemplos

- location.href -> Nos devuelve la dirección URL. Tambien nos sirve para cambiar el valor de la URL, ejemplo: location.href = "https://google.com"
- location.protocol -> Nos devuelve el protocolo de la pagina (http, https, etc).
- location.host -> Nos devuelve el dominio principal que estamos visualizando. A pesar de estar en una carpeta o un subdirectorio, nos da la localización principal del dominio siempre.
- location.pathname -> Lo contrario a host. Nos da la localización de donde estemos en el dominio.
- location.hash -> Es una forma de pasar parametros entre pagina, podemos utilizarlo para tener un solo HTML y en función del hash que pasemos cargar una información u otra.
- location.reload() -> Recarga la pagina.

Objeto history - Es el objeto que trabaja con el historial de la pestaña donde navegamos.

https://developer.mozilla.org/es/docs/DOM/Manipulado_el_historial_del_navegador

- back() -> Va hacia la pagina web anterior de la pestaña (Si existe).
- foward() -> Va hacia la pagina web posterior de la pestaña pestaña (Si existe).
- go(n|-n) -> Positivo va a X cantidad posterior, negativo va -X cantidad anterior (Si existe).
- length -> Nos da la cantidad de paginas web que se abrieron en la pestaña.

Objeto date - Es el objeto permite trabajar con fechas y horas.

https://developer.mozilla.org/es/docs/JavaScript/Referencia/Objetos_globales/Date

https://www.w3schools.com/jsref/jsref_obj_date.asp

Timers - Los timers nos permiten establecer que una función se ejecuta despues de X tiempo y automaticamente al leerla.

Timeout:
https://developer.mozilla.org/en-US/docs/Web/API/WindowOrWorkerGlobalScope/setTimeout

- setTimeout(() => {code}, delay in miliseconds) - Hace que se ejecuta la función después del delay. Si lo referenciamos mediante una variable/constante podemos pararlo con clearTimeout(referencia).

Interval:
https://developer.mozilla.org/en-US/docs/Web/API/WindowOrWorkerGlobalScope/setInterval

- setInterval(() => {code}, delay in miliseconds) - Hace que se ejecute una función cada delay milisegundo. Si lo referenciamos mediante una variable/constante podemos pararlo con clearInterval(referencia).

4. Peticiones HTTP

AJAX

AJAX, acrónimo de Asynchronous JavaScript And XML, es una técnica de desarrollo web para crear aplicaciones web asíncronas. Estas aplicaciones se ejecutan en el cliente, es decir, en el navegador de los usuarios mientras se mantiene la comunicación asíncrona con el servidor en segundo plano.

Hay que leer la documentacion de la API para saber como el backend implementa los metodos HTTP y saber de que manera interactuar con el servidor para recibir, enviar, actualizar, etc. información.

Por ejemplo: https://jsonplaceholder.typicode.com/guide/ -> Sabemos que con el metodo GET vamos a recibir informacion, aplicando el POST de la manera que indica la documentación vamos a enviar información a un servidor y lo va a cargar, etc.

- Pequeña definición de AJAX

¿Cómo funciona la Web? Nosotros hacemos una petición a través del protocolo HTTP a un dominio, solicitando a un servidor o una nube esa cierta información. Ese servidor nos devuelve esa información como página web, que esta compuesta por HTML, CSS, JS, imagenes, video, etc. Así es como funcionan todas las webs que no utilizan AJAX.

Cuando nosotros utilizamos AJAX lo que estamos haciendo es interceptar esa respuesta para evitar que la pagina se recarge. De esta forma lo que hacemos es, en lugar de solicitar todos los datos, hacemos la petición a través del protocolo HTTP pero solo solicitamos ciertos datos, y esos datos cuando llegan lo que hacemos es guardarlos en un objeto. Entonces de esta forma obtenemos los datos del servidor pero no recargamos la pagina porque no estamos cargando esos datos en el navegador, los tenemos en un objeto y una vez que esta ahi trabajamos con ese objeto de la forma que necesitemos.

Otra definicion: https://es.wikipedia.org/wiki/AJAX

Las peticiones a una API se ven normalmente como objetos construidos en formato JSON.

- API

https://www.redhat.com/es/topics/api/what-are-application-programming-interfaces

Una API o interfaz de programación de aplicaciones es un conjunto de definiciones y protocolos que se usa para diseñar e integrar el software de las aplicaciones.

Las API permiten que sus productos y servicios se comuniquen con otros, sin necesidad de saber cómo están implementados. Esto simplifica el desarrollo de las aplicaciones y permite ahorrar tiempo y dinero. Las API le otorgan flexibilidad; simplifican el diseño, la administración y el uso de las aplicaciones; y ofrecen oportunidades de innovación, lo cual es ideal al momento de diseñar herramientas y productos nuevos (o de gestionar los actuales).

Otra definicion: https://es.wikipedia.org/wiki/Web_API#:~:text=Una%20API%20es%20una%20interfaz,funciones%20de%20un%20determinado%20software.

En este curso no veremos AJAX con JQuery al ser una libreria pesada y obsoleta. Veremos como hacer peticiones sin JQuery y que sean compatibles con navegadores antiguos.

Normalmente las peticiones a los servidores se hacen a través de PHP, es la forma más común. Hoy por hoy node esta comiendo parte del mercado, sin embargo PHP sigue siendo la principal.

Vamos a utilizar la API https://jsonplaceholder.typicode.com para realizar peticiones AJAX, ademas de montar un pequeño servidor PHP (WampServer).

1. Objeto XMLHTTPRequest

Para guardar/lanzar una peticion por convencion utilizamos el nombre xhr para guardar el objeto XMLHttpRequest.

Para saber si un navegador soporte un objeto XMLHttpRequest simplemente preguntamos en un if si exisite el objeto en la clase window. Si existe asignamos la variable xhr a un new de XMLHttpRequest(), de lo contrario utilizamos el objeto ActiveXObject("Microsoft.XMLHTTP");

De esta manera nos ahorramos todo jQuery.

La variable en la cual guardamos el objeto XMLHttpRequest/ActiveXObject("Microsoft.XMLHTTP"); tiene un metodo llamado open(). Este metodo recibe en su primer parametro el metodo que vamos a utilizar para hacer la peticion (puede ser GET, POST, PUT, DELETE), y en su segundo parametro el link interno u externo al cual queremos hacer la conexion para la peticion.

Aclaracion: Si se hace a través de GET los codigos/parametros (por ejemplo los que aparecen en un metodo GET de formularios o el ejemplo) a acceder hay que ponerlos en la URL a al que hacemos la peticion. En cambio por POST tendriamos que crear un objeto donde vamos a guardar esa información.

Podemos hacer peticiones para enviar datos a un lugar o recibir datos de ese lugar. Por ejemplo un GET a un JSON como esta ahi debajo, enviar datos en JSON a una pagina o cargar datos a una DB.

Ejemplo del metodo POST en la carpeta "1.3 Objeto XMLHTTPRequest 3"

Ejemplo accediendo a una ruta externa:

- xhr.open("GET", "https://jsonplaceholder.typicode.com/users");

Con el metodo send() enviamos esta petición.

Ejemplo:

- xhr.send();

Sin embargo, antes de enviar esta peticion debemos decirle que hacer con esos datos antes de enviarlos. Por lo tanto antes del send debemos codear que vamos a hacer con la información recibida. También debemos verificar que el hilo asincrono haya recibido toda la información antes de enviarla/procesarla, para ello utilizamos el evento load. Este evento se dispara cuando toda la información a llegado al objeto.

Ejemplo parseando del JSON recibido solamente el ID y nombre:

```
xhr.addEventListener("load", (data) => {
const dataJSON = JSON.parse(data.target.response);
const list = document.getElementById("list");
for(let userInfo of dataJSON){
   const listItem = document.createElement("li");
   listItem.textContent = `${userInfo.id} - ${userInfo.name}`
   list.appendChild(listItem);
}
```

```
});
```

2. Callbacks

Un callback es una función que se ejecuta a través de una función. Los callbacks se volvieron obsoletos ya que son dificiles de mantener y se empezaron a utilizar las promesas. Los callbacks no son asincronos.

3. Promesas

Documentación:
https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Using_promises

Una promesa es un objeto que dentro tiene dos callbacks. Es un objeto que representa la terminación o el fracaso de una operación asíncrona.

Otra definicion

En promesas utilizamos funciones de tipo callback.

Una promesa básicamente es código que tiene varios estados, así que vamos a poder lanzar una petición para procesar un código. En dado caso que la promesa se resuelta correctamente se resuelve el codigo y en caso que haya tenido problemas se manda a llamar el error.

Por lo tanto podemos decir que existen dos caminos: Un en el que se ejecuto el codigo correctamente y se resolvio el mismo y otro en el cual falla el codigo y se manda a llamar el error en el catch.

Mientras la promesa no haya terminado de ejecutar el codigo se encuentra en estado de pendiente. Una vez ejecutado el codigo si se puede resolver utilizamos la funcion .then(), en caso de que no se haya podido procesar utilizaremos la funcion .catch().

Una Promise (promesa en castellano) es un objeto que representa la terminación o el fracaso de una operación asíncrona. Dado que la mayoría de las personas consumen promises ya creadas, esta guía explicará primero cómo consumirlas, y luego cómo crearlas.

La palabra async nos va a permitir el uso de promesas. Al poner la palabra antes de la definición de un método significa que esta obligado a regresar una promesa.

La palabra reservada await solo puede utilizarse dentro de una función aysnc.

Una promesa se crea con el constructor de la clase Promise(FUNCION); Dentro de los parentesis recibe como argumentos un puntero a función o una función anonima. Ambas funciones deben tener un parametro de resuelto y otro de rechazado, ambos parametros son callbacks.

Ejemplo: const promise = new Promise((resolve,reject) => {});

Una vez resuelta la función se devuelve el objeto creado como Promesa(). Nos devuelve el estado fulfilled si pudo terminar su ejecución.

Para leer el contenido de las promesas tenemos los metodos .then() si el retorno de la promesa fue resolve y .catch() si ocurrio algún error y retorno rejected. Ambas funciones reciben punteros a funcion o funciones anonimas.

Los catch funcionan para todos los then() que utilicemos. Los then() no hacen falta ser anidados.

4. Fetch - Introducción

Es el reemplazo moderno del objeto XMLHTTPRequest.

Fetch API

Proporciona una interfaz JavaScript para acceder y manipular partes del canal HTTP, como peticiones y respuestas. Básicamente y muy por encima lo que hace es encapsular y crear una promesa con sus callbacks.

También provee de un método global fetch() que proporciona una forma fácil y lógica de obtener recursos de forma asíncronica por la red.

Esta basado en Promesas, por lo cual tiene un response y un reject internos.

- Response tiene varios métodos:
    - arrayBuffer(): Archivos binarios en bruto (mp3, pdf, jpg, etc). Se utiliza cuando se necesita manipular el contenido del archivo.
    - blob(): Archivos binarios en bruto (mp3, pdf, jpg, etc). Se utiliza cuando no se necesita manipular el contenido y se va a trabajar con el archivo directamente.
    - clone(): Crea un clon de un objeto de respuesta, idéntico en todos los sentidos, pero almacenado en una variable diferente.
    - formData: Se utiliza para leer los objetos FormData.

- json(): Convierte los archivos json en un objeto de JavaScript.
- text(): Se utiliza cuando queremos leer un archivo de texto. Siempre se codifica en UTF-8.

Fetch por defecto al ponerle una URL trabaja con el metodo GET
fetch("RUTA").then(MANEJO SI NO HUBO ERROR).catch(MANEJO SI HUBO ERROR);

Ejemplo:

```
fetch("https://jsonplaceholder.typicode.com/users")

.then(res => res.ok ? Promise.resolve(res) : Promise.reject(res)) //Si resolvio procede a los then, sino hace el reject y pasa al catch.

 .then(res => res.json())

.then(res => {

   const list = document.getElementById("list");

   const fragment = document.createDocumentFragment();

   for(const userInfo of res){

      const listElement = document.createElement("li");

      listElement.textContent = `${userInfo.id} - ${userInfo.name}`;

      fragment.appendChild(listElement);

   }

   list.appendChild(fragment);

})

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

5. Fetch - Peticiones POST

Fetch API

Las peticiones POST sirven para enviar datos y hacer incersiones en una API o en una base de datos o en algun sitio.

Para hacer peticiones POST, fetch admite un segundo parámetro.

Ejemplo básico de peticion POST.

```
fetch(url,{
    method: 'POST',
    body: Los datos que enviamos. Si es un objeto hay que convertirlo con JSON.stringify(datos),
    headers: {
        Cabeceras de información sobre lo que estamos enviando.
        https://developer.mozilla.org/es/docs/Web/HTTP/Headers
    }
})
```

6. Fetch - Lectura de archivos

Para leer archivos sin manipular su contenido debemos generar un Binary Long Object (blob) del asset que estamos adquiriendo. Lo convertimos en binario para que se puede leer en todos los dispositivos.

Ese blob para que pueda ser visualizado lo vamos a convertir en un objeto URL con la funcion URL.createObjectURL(blob asset).

Ejemplo:

```
.then(asset => asset.blob())
.then(asset => {
    document.getElementById("img").src = URL.createObjectURL(asset);
})
```

7. async/await

Llego con el estandar EMAC Script 7. Es una implementación para hacer funciones asíncronas en JavaScript de forma nativa.

Las funciones asincroncas devuelven promesas. Podemos declarar dentro una Promise y JS va a detectar que tenemos una promesa. Evitando así que de devuelve una promesa de una promesa.

async crea una función asincrona, esta funcion asincrona no va a bloquear el event loop/stack y cuando termine se va a mostrar el codigo de la funcion async. Si tenemos que recibir informacion que puede llegar a tardar varios segundos es importante aplicar este concepto para que la página no se cuelge.

Si una función depende de otra asincrona ésta función debera ser asincrona también (ya que al depender de otra asincronia no se ejecuta en secuencia) y al llamar a la función asincrona original ponemos un await delante de la función para indicarle que espere a que termine el proceso y después continue con el código restante.

Se podría decir que await bloquea el en el stack mi funcion hasta que se resuelva la otra funcion del stack. Si no bloqueramos la funcion que llama a mi función principal asincrona es probable que nos devuelva un object vacio en vez de la información que recaba el hilo asincronico.

await no puede funcionar si no estamos dentro de una función async.

- Ejemplo promesa explicita:

```
const getName = async () => {
   return new Promise((resolve, rejected) => {
      setTimeout(() => {
         resolve("Luciano");
      }, 1000);
   })
}
```

```
const saludar = async() => {
    const nombre = await getName();
    return `Hola ${nombre}`;
}

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

- Ejemplo promesa implicita

```
const users = [{id:1,name:"Pedro"}, {id:2,name:"Laura"}, {id:3,name:"Carlos"}];
const emails = [{id:1,email:"Pedro@mail.com"}, {id:2,email:"Laura@mail.com"}];

const getUser = async (id) => {
    const user = users.find(user => user.id === id);

    if(user == undefined){
        throw new Error(`Not exist a user with ID ${id}`);
    }

    return user;

}

const getEmail = async (user) => {
    const email = emails.find(email => email.id === user.id);

    if(email == undefined){
        throw(`Not exist a email with ID ${user.id}`);
    }
```

```
    return ({
        id: user.id,
        name: user.name,
        email: email.email
    });
}

const getInfo = async(id) =>{
    try{
        const user = await getUser(id);
        const res = await getEmail(user);
        return `${res.name} el mail es ${res.email}`
    }catch(error){
        console.log(error);
    }

}

getInfo(1).then(res => console.log(res));
```

O si no utilizamos try catch.

```
getUser(2).then(res=> getEmail(res)).then(res =>
console.log(res)).catch(error=>console.log(error));
```

8. Libreria AXIOS - GET/POST

Documentacion: https://github.com/axios/axios.git

Axios es una libreria basada en promesas tanto para el cliente como para el servidor que utiliza JSON. Es una libreria que convierte automaticamente a json las peticiones y reemplaza a jQuery, fetch, entre otras cosas.

5. Formularios y expresiones regulares

1. Formularios - Introducción a validación de formularios

Es importante, por mas que este el boton deshabilitado, prevenir el comportamiento default a la hora de enviar un formulario. Si una persona tiene conocimiento de HTML puede deshabilitar esta opción en la consola del navegador y enviar el formulario antes de validarlo tanto en el frontend como en el backend.

Por eso es importante asegurarse que el script de JS contenga este tipo de incidencias.

Si tenemos un boton para enviar es importante no utilizar el id "submit" ya que genera conflictos.

Para acceder a un valor dentro de un formluario tomamos el id del formulario con un getElementById y accedemos a traves del operador '.' poniendo el name/id que le dimos a la etiqueta.

You can access a particular form control in the returned collection by using either an index or the element's name or id.

https://developer.mozilla.org/en-US/docs/Web/API/HTMLFormElement/elements

2. Introduccion a expresiones regulares

Pagina para test: https://regex101.com/

Documentacion: https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/RegExp

Son secuencias de caracteres que forman un patrón de búsqueda, principalmente utilizada para la búsqueda de patrones de cadenas de carecteres u operaciones de sustitución.

Sintaxis:

- /patron/banderas

Banderas:

- i: ignore case. No diferencia entre mayusculas y minusculas.
- g: global. Busca la forma global, es decir, no se para después de la primera coincidencia.

Ejemplo de utilizacion:

- const texto = document.getElementById("texto");
- const regEx = /lorem/gi; -> Genero la expresion regular implicitamente.
- console.log(regEx.test(texto.textContent)); -> El text devuelve true y hay coincidencia, false si no hay.

3. Expresiones regulares complejas

Sintaxis:

- /patron/banderas

Banderas:

- i: ignore case. No diferencia entre mayusculas y minusculas.
- g: global. Busca la forma global, es decir, no se para después de la primera coincidencia.

Comodines:

- Sustitución: Define un comodin dentro del patrón. El símbolo es el ".". Si sabemos ciertas palabras dentro del patron pero otras no utilizamos el . que toma cualquier caracter. i..um toma tanto "ipsum" como "i$%um".
- Listado de caracteres válidos: Entre corchetes se pona una lista con los cáracteres válidos. [aeiou] Con esto tenemos todas las vocales.
- Rangos: Entre corchetes si pones un guión entre dos caracteres establecemos un rango, utiliza la tabla ASCII. [a-z] Todas las letras minúsculas.

- Mezcla entre rangos y listas: Podemos unir los dos anteriores en una sola expresión. [0-5ou] Serían números del 0 al 5, la letra "o" y la letra "u".
- Cadenas completas: Para establecer una cadena completa debe ir entre paréntesis, si queremos más palabras irán separadas por una pipe. (lorem|amet) es válida para la palabra "lorem" y la palabra "amet".
- Escapar un caracter: Para que la expresion regular reconozca un caracter dentro de la cadena utilizamos \ anterior al caracter para que se lo reconozca como tal. Como en C en una cadena de caracteres.

Delimitadores:

- ^ Antes de este símbolo no puede haber nada.
- $ Después de este símbolo no puede haber nada.
- ^hola$ -> Antes de la h nada, despues de la a nada.

Cantidad:

- Llaves: Lo que está antes tiene que aparecer la cantidad exacta de veces. Hay tres combinaciones posible.
- {n} Se tiene que repetir n cantidad de veces.
- {n, m} Se tiene que repetir entre n y m veces, ambas incluidas.
- {n, } Se tiene que repetir como mínimo n veces.
- ^[a-zA-Z]{1,3}@{1}$ -> 1 a 3 letras con un @ obligatorio.
- Asterisco: Lo que está antes del asterisco puede estar, puede no estar, y se puede repetir. .*@*\..*
- Interrogación: Lo que está antes de la interrogación puede no estar, pero si está solo puede aparecer una vez. ^[ae]?$
- Operador más: Lo que está antes del + tiene que estar una vez como minimo A-[0-9]+

Caracteres:

- \s: Coincide con un cáracter de espacio, entre ellos incluidos espacio, tab, salto de página, salto de línea y retorno de carro. ^[a-zA-Z]+\s[a-zA-Z]+$
- \S: Coincide con todo menos caracteres de espacio ^\S{5}$.

- \d: Coincide con un cáracter de número. Equivalente a [0-9] ^\d{5}$
- \D: Coincide con cualquier cáracter no númerico. Equivalente a [^0-9] ^\D{5}$
- \w: Coincide con cualquier cáracter alfanumérico, incluyendo el guión bajo. Equivalente a [A-Za-z0-9_] ^\w+@$
- \W: Coincide con todo menos caracteres de palabra. ^w+@$

Declaramos una expresion regular de manera explicitamente con el contructor RegExp("PATRON/COMODINES", "BANDERAS");

Dos ejemplos:

```
const regEx = RegExp("lorem", "gi");

const regEx = RegExp("/lorem/", "gi");
```

6. API's HTML

1. Web Storage

Hasta la aparicion de los primeros dispositivos moviles la informacion se guardaba en un servidor porque no habia otra forma de guardarla. Después surgio el tema de las cookies que nos permitia guardar informacion en el equipo pero se quedaban limitadas. Las cookies permiten guardar informacion pero hasta cierto punto, eran en formato texto y de tamaño limitado.

La API de WebStorage nos permite guardar informacion en el dispositivo con el conjunto de clave:valor con un limite de tamaño mucho mayor que las cookies.

Los dos mecanismos en el almacenamiento web son los siguientes:

- sessionStorage mantiene un área de almacenamiento separada para cada origen que está disponible mientras dure la sesión de la página (mientras el navegador esté abierto, incluyendo recargas de página y restablecimientos).
- localStorage hace lo mismo, pero persiste incluso cuando el navegador se cierre y se reabra.

Ambos funcionan con clave:valor y tienen dos métodos fundamentales setItem() para asignar una clave:valor y getItem() que recibe como parámetro la clave de la que quieremos obtener el valor.

Podemos acceder al Almacenamiento local/sesion en la pestaña de Aplicacion en Chrome.

Para eleminar contenido en ambos casos utilizamos .clear() para borrar todo y removeItem(KEY) para remover un item.

2. Drag and Drop

Esta API nos permite arrastrar y soltar cosas dentro del navegador

Existen dos zonas principales en esta API, el objeto a arrastrar y la zona donde vamos a dejarlo.

Para controlar estas acciones tenemos varios eventos de cada una de las partes

- Objeto a arrastrar:
  - dragstart: Se dispara al comienzo de arrastrar.
  - drag: Se dispara mientras arrastramos.
  - dragend: Se dispara cuando soltamos el objeto.
- Zona destino:
  - dragenter: Se dispara cuando el objeto entra en la zona de destino.
  - dragover: Se dispara cuando el objeto se mueve sobre la zona de destino.
  - drop: Se dispara cuando soltamos el objeto en la zona de destino.
  - dragleave: Se dispara cuando el objeto sale de la zona de destino.

El atributo draggable="true" dentro de un div nos da la posibilidad de poder arrastrarlo.

Hay que utilizar un objeto llamado dataTransfer. Contiene toda la informacion del objeto que estamos arrastrando. Con la propiedad files podemos acceder a archivos si estamos arrastrando/ingresando archivos.

Existen dos metodos:

setData(FORMATO, DATA): Establece la información que queremos compartir

getData: Establece la información que queremos obtener

3. API File

Esta API nos sirve para leer archivos que el usuario cargue en la web. Se pueden cargar archivos desde un input de tipo file o desde el objeto DataTransfer de la API Drag&Drop.

La interfaz más utilizada para interactuar con ella es FileReader

https://developer.mozilla.org/es/docs/Web/API/FileReader

El atributo multiple dentro del input nos permite seleccionar mas de un arhchivo.

Cuando lanzamos el evento change la propiedad e.target.file nos devuelve todos los archivos seleccionados en la interfaz.

4. API IndexedDB

Especificación oficial: https://developer.mozilla.org/es/docs/IndexedDB-840092-dup

IndexedDB es una base de datos indexada y es una manera de almacenar datos de manera persistente en el navegador. La diferencia principal entre IndexedDB y Web Storage es que ésta ultima sirve para porciones muy pequeñas de datos

y en IndexedDB permite guardar una gran cantidad de datos.

Almacena pares llave:valor. Los valores pueden ser objetos y estructuras complejas, y las llaves pueden ser propiedades de esos objetos.

La API IndexedDB es mayormente asíncrona.

IndexedDB usa eventos DOM para notificar cuando los resultados están disponibles.

IndexedDB es orientada a objetos.

- Creación

Crear la base de datos a través del objeto indexedDB y el método open('Nombre', Version_INT).

Comprobar si la base de datos existe o tiene que ser creada a través del método onupgradeneeded(). Si no existe la crea.

Crear almacén de objetos con el método createObjectStore().

Escuchar los eventos de éxito y de error con los métodos onsucces() y onerror().

Es importante a la hora de crear un objectStore generar claves autoincrementales para poder hacer un push de data sin keys asociadas.

- Ejemplo: db.createObjectStore('tasks', { autoIncrement: true });

En el autoincrement podemos generar otros tipos de claves mas especificas.

- Añadir datos

Todas las operaciones sobre una base de datos indexada funcionan a través de una transacción. Una transacción recibe 2 parámetros, el almacen donde vamos a trabajar y de que modo vamos a trabajar.

- Ejemplo: db.transaction(['tasks'], 'readwrite');

Una vez tenemos la transacción creada vamos a abrir el almacen de datos.

- Ejemplo: const objectStore = transaction.objectStore('tasks')

Y luego añadimos el objeto a la db. objectStore.add(data);

5. API Visibility Change

Con la propiedad document.visibilityState podemos saber si nuestra pagina esta siendo visualizada o no. La propiedad devuelve visible o hidden.

Nos puede ser muy util para parar videos o reproducciones de audio y ahorrar recursos.

El evento asociado es visibilitychange

- Ejemplo aplicación: addEventListener('visibilitychange', () => document.visibilityState === 'visible' ? video.play() : video.pause())

6. API Online/Offline

Este evento nos permite saber si el navegador esta online u offline con respecto a la conexion a internet.

Esta API esta en la clase windows. Por lo tanto solamente vamos a tener que agregar el addEventListener.

Los eventos asociados son 'online' y 'offline'.

7. API Intersection Observer

https://developer.mozilla.org/es/docs/Web/API/Intersection_Observer_API

Esta API sirve para saber cuando un elemento se ve y cuando esta oculto. Sirve mucho para hacer lazy load/carga perezosa, es decir, no cargamos elementos que no vemos. Un ejemplo de esto es Twitter o Instagram. Cargan 10/15 imagenes y cuando se comienza a hacer scroll pide cargar mas imagenes.

8. API Geolocation

https://developer.mozilla.org/es/docs/Web/API/Window/matchMedia

mql = window.matchMedia(mediaQueryString)

mql viene de media query list, que es el objeto que se crea con el método matchMedia()

mediaQueryString es cualquier media query válida en CSS

7. Destructing, debugging y transpilacion

1. Destructing

https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Operadores/Destructuring_assignment

La destructuración es una expresión de JavaScript que hace posible la extracción de datos de arreglos u objetos

8. Shortcuts

Pagina: https://medium.com/@ashley.karhoff1/emmet-shortcuts-to-quickly-generate-html-tags-and-css-classes-aeb04cd48aec

Shortcut para crear varios elementos HTML duplicados: element{EJEMPLO-$}*CANTIDAD -> Las {} son opcionales, el $ da comienzo a partir del 1. Es posible combinarlo al final con el "." o el "#".

Para anidar elementos utilizamos el simbolo ">": Ejemplo: nav>ul>li*4>a

Añadir un div (por defecto) con una clase: ."nombreClase" -> Poniendo puntos añado mas clases

Añadir un div (por defecto) con un id y una clase (opcional): #"nombreId"."nombreClase" -> Poniendo puntos añado mas clases