**JAVASCRIPT DESARROLLADOR AVANZADO**

**discor profe** [**https://discord.gg/MtVU8BJd**](https://discord.gg/MtVU8BJd)

**JavaScript (JS) es un lenguaje ligero e interpretado, orientado objetos con funciones de primera clase, más conocido como el lenguaje de script para páginas web, pero también usado en muchos entornos sin navegador, tales como node.js o Apache CouchDB. Es un lenguaje script multi-paradigma, basado en prototipos, dinámico, soporta estilos de programación funcional, orientada a objetos e imperativa.**

**https://www.freecodecamp.org/espanol/news/aprende-javascript-imports-exports-let-const-promisas-es6/**

**CONSTRUCTORES DE VARIABLES ESC6**

**Una funcion Global es aquella que tiene alcance fuera del bloque de codigo no funcional en que se encuentra.**

**VAR:** Variable GLOBAL que puede ser redefinida y modificada desde cualquier bloque de codigo no funcional.

- Redeclarables

- Redefinibles

- Alcance global

**LET**: Puede redeclararse en ambito local pero no global. Si puede ser redefinida en ambito local. Tambien puede ser redeclarada localmente en una funcion.

- NO redeclarables

- Redefinibles

- NO globales

**CONST**: No puede ser redefinida ni modificada. Tambien se puede recrear localmente en un funcion. Tienen alcance dentro del bloque en que se crearon

- NO redeclarables

- NO Redefinibles

- NO globales

- Admiten redefinicion de componentes internos. Ej los elementos de un objeto construido con const

**DATOS PRIMITIVOS ( pasan sus datos por valor, hacen una copia del valor que tienen dentro )**

**string:** “string”

**numero:** 1234

**booleano: t**rue / false

**null: valor** null

**undefined:** propiedad de alto nivel con valor no definido.

**Symbol:** simbolos

**DATOS OBJETO ( pasan sus datos por referencia, siempre se mantiene una referencia viva )**

**Array:** Matriz ordenada de valores en manera ascendente numerica ascendente.

**Objeto:** Matriz ordenada asociativa de indice: valor

**Función:** Objeto con la habilidad de poder ser ejecutados.

**ACCESO A OBJETOS const obj = { x:1, y:2, 0:true }**

**Notacion de corchetes**

Indice numérico **obj[0]**  // true

indice string **obj[“x”]** // 1

indice variable **const indice = “x” / obj[indice]** // 1

**Notacion de puntos**

Indice numérico **obj.0**  // NO permitido usando numeros

indice string **obj.x** // 1

indice variable **const indice = “x” / obj.indice** // NO posible. Arroja undefined

**ITERACIONES**

**Metodo for each:** Especial para recorrer arrays, es una Funcion callback sincronica que se ejecuta por cada elemento del array.

**Metodo for in:** Especial para recorrer objetos ( matrices asociativas )

**FUNCIONES LAMBDA O FLECHA**

**Siempre son**

**- FUNCIONES ANONIMAS**

**- NO relacionadas con metodos**

**- NO pueden utilizarse como CONSTRUCTORES.**

**FUNCION ANONIMA:** Funcion sin nombre que se asigna a una variable.

**let hola = function() { console.log(“hola”) }**

**let hola = () => { console.log(“hola”) }**

**let hola = parámetro => { console.log(“hola”) }** // si solo recibe un parametro se omite el ( )

**FUNCION FLECHA**: Se omite la palabra function. Si tiene un solo parametro se omiten los ( ), si el cuerpo tiene una sola línea se omiten los { }, y si no se especifica tiene un retorno implícito (no hace falta escribir return.

ej: let doble = a => 2 \* a // retorna 2\*a

**TEMPLATE STRINGS**

**FUNCION ANONIMA:** Se imprime usando los back ticks y se imprime tal cual se escribe dentro. A las variables se las llama con ${var}

ej: console.log(`Hola que tal ${nombre} ${apellido}`) // Imprime “Hola que tal nombre apellido”

**BOM Y DOM**

**BOM ( browser objet model ) :** Es una representacion global del navegador recreada por javaScript. Es un modelo del navegador, en forma de objeto JSON, escrito por Js. Esta variable se encuentra dentro de la referencia **window**, y mantiene una referencia viva de todo lo que hagamos en js y todo a lo que podamos acceder y manejar desde el navegador.

Ejemplos de propiedades del objeto window

window.innerHeight // Alto del documento abierto  
window.innerWidth // Ancho del documento abierto  
 //Los inner\* no toman en cuenta la consola de desarrollo, barras de scroll, barra de navegación, etc.

window.outerHeight // Alto de todo el navegador  
window.outerWidth // Ancho de todo el navegador  
 //Los outer\* toman en cuenta absolutamente todo en el navegador

window.location.href // Variable que controla la direccion de la barra de navegacion

window.console // Objeto interfaz de la API Console. En el tenemos la popular función log

window.alert // Para notificaciones de alerta  
window.confirm // Para ventanas de confirmación  
window.prompt // Para ventanas de ingreso de texto  
window.history // Para revisar el estado de historial de navegación del cliente  
window.document // Para acceder a información relativa  
del documento abierto

**DOM ( document objet model ) :** Es la manera en que jscript representa HTML. Es una api web que en forma de objeto Js hace una representación del documento que tengamos abierto. Es una api web que está incluida en el objeto window y se accede a través de la propiedad document, donde podemos visualizar y acceder a todas las propiedades y metodos disponible para utilizar dentro de html.

Ejemplos de propiedades del objeto document

let h1 = document.createElement("h1")  
  
document.getElementById(id);  
document.getElementsByClassName(names);  
Element.getElementsByTagName(name);  
Element.querySelector(selectors8);  
Element.querySelectorAll(selectors)

**MANIPULACION DEL DOM**

**EDITAR UN NODO:**

Cada nodo, desde document en adelante, va a hacer referencia a una etiqueta HTML. Los atributos de HTML de cada nodo se verán reflejados como sus propiedades. Los atributos de HTML con su valor inicial se almacenan en la propiedad  **attribute**.

**Para manipular correctamente el body desde un script, es conveniente relacionar el script desde el final del body, sino el navegador leera el script antes de la construccion del body y no lo podra aplicar.**

**<body>**

**//codigo**

**<script src=”script.js”> <script>**

**</body>**

**SELECTORES de HTML:** Afectamos a elementos html ya creados.

**- document.getElementById(‘nombre\_Id’)**

**- document.getElementByClassName(‘nombre\_Clase’)**

**- document.getElementByTagName(‘tipo\_etiqueta’)**

**- document.querySelector(‘#titulo’) /**/ selector por id

**- document.querySelector(‘.titulo’)** // selector por clase

En los casos de selección múltiple como Clase y Tag, java Script creará un array con todos los elementos porque estos son valores que podrían repetirse. Si tengo mas de un elemento en el array puedo acceder a uno especifico espacificando su ubicacíon dentro del array. Ej:

**- document.getElementByTagName(‘h2’)[1]**

Mediantes Js podemos manipular y editar todas las propiedades de los elementos seleccionados del DOM.

**CREADORES de ELEMENTOS HTML:** Afectamos a elementos html ya creados.

**- var p = document.createElement(‘p’) // crea un elemento con el nombre de una etiqueta valida y lo guarda en memoria.**

**- document.body.appendChild(p) // asigna el elemento creado al body del DOM**

**- document.removeChild** **//remueve un nodo existente.** Ej: div.removeChild(‘p’)

**MANIPULAR CLASES:** Las clases se guardan en una propiedad **className** dentro de un string. Para manipular clases js ofrece facilidades como:

h1.classList.add(“titulo”)

h1.classRemove(“titulo”)

h1.class.toggle(“titulo”)

**MANIPULAR PROPIEDADES:** Las clases se guardan en una propiedad **className** dentro de un string. Para manipular clases js ofrece facilidades como:

let h1 = document.querySelector(“#tituloh1”)

h1.style.color = ‘orange’

podemos ver todas las propiedades disponibles de un elemento del dom seleccionado, si en la consola escribimos el nombre de la variable que almacena el elemento con el punto al final. Ej h1.

**DOCUMENT FRAGMENT:** Mediante document fragment creamos un DOM temporal en el cual incluimos varios cambios sin aplicar, que luego podrán ser incormporados de una sola vez mediante el document fragment y despues lo asignamos al body.

Ej const dfrag = documet.createDocumentFragment()

body.appendChild(dfrag)

**CALLBACK**

**Callback**

Un callback es una función que se envía como parámetro para ejecutarse desde otra función. El callback se envía como parámetro sin los ( ).

**EVENTOS**

**Evento**

Un evento es la ejecucion de una función como respuesta a una accion. Cada evento está representado por un Objeto que se basa en la interfaz **Event**. Los desencadenantes de eventos incluyen la finalizacion de la carga de un recurso en la red. Actualmente todos los entornos de ejecucion para codigo JS usan eventos y manejo de eventos.

**ARQUITECTURA DE UN EVENTO**

\* Nombre del string utilizado para el evento

\* Tipo de estructura de datos utlizada para representar las propiedades claves de ese evento.

\* El ojeto JS que ‘emitira’ ese evento.

Este patron es implementado por:

\* Una funcion de JS que toma como argumento la estructrua de datos que se acordó

\* registrando la funcion con un string con el objeto que emitira el evento.

**Definiendo eventos**

Un evento impacta sobre elementos del HTML. Un atributo puede ser manipulado mendiante propiedades incluidas como atributos de los emelmentos HTML, directamente desde el HTML, pero no es lo ideal ya que cada funcionalidad deberia ser escrita en su propio lenguaje, y ademas posee varias limitaciones.

**Eventos desde html** *Ej <button onclick=”console.log(‘click’)>Boton Click</button>*

error ( aplicar la propiedad onclick desde js *btn.onclick = console.log(‘click’)* resultaria en un error ya no estamos enviando un callback para ser ejecutado postriormente sino que cuando carguemos la pagina el navegador va a ejecutar la funcion console.log que trae asociada la variable btn automaticamente en la linea que esté escrita.

**Eventos html desde JS con callback** *Ej function uno () { console.log(‘uno) }*

*btn.onclick= uno()*

De esta manera asignamos un callback a la variable btn, que trae como refetencia el boton del html con su propiedad onclick. Esto es valido pero el problemas es que le estamos asignando un callback a una variable, por lo que solo le podemos asignar un unico callback, si volvemos a reasignarle otro, tendremos que reescribir el valor de la variable y el nuevo callback pisará al anterior.

**API web EventTarget:**

Esta api web nos permite registrar eventos en multiples elementos ya sean BOM, DOM y cualquier ELEMENTO html.

Para esto, la Api nos provee el metodo **addEventListener**

**element*.addEventListener (‘evento’, callback1 ) //***

**element*.addEventListener (‘evento’, callback2 )***

***- elemento***

***- addEventlListener***

***- evento en nombre de string //***  *parametro obligatorio*

***- funcion callback a ejecutar*** *// parametro obligatorio*

**EL OBJETO EVENT**

Es un parametro que se pasa como callback por defecto.

**element*.addEventListener (‘evento’, funcion(e) {***

***console.log(e)***

***})***

Si en vez de pasar como callback una funcion ya definida, enviamos una funcion anonima con el objeto evento como parametro, esto nos permite acceder a todas las funcionalidades del objeto Event.

**PROPAGACION DE EVENTOS**

Los Eventos em JS siempre ( por defecto ) se propagan desde adentro hacia afuera, es decir desde los elementos hijos a los padres. El motor de JS revisara todos los nodos en forma ascendente, y si encuentra en aglún nodo un evento del mismo tipo que el evento disparador, lo disparará también.

**Utilidades de la propagacion de Eventos. // ELEMENTOS DINAMICOS**

Es de suma utilidad para asignar eventos a elementos **dinamicos,** es decir a elementos que no fueron declarados estáticamente en el html.

Por ejemplo, si quisieramos asignar un evento a un elemento dinamico, no podriamos directamente con **element.addEventListener** porque no podriamos acceder al mismo, porque el DOM no lo reconoce como elemento.

Entonces, podemos asignar un listener al elemento mayor, y encontrar mediante la propiedad .**target** del **objeto e,** el disparador del evento y mediante la propiedad **target** acceder a su **id, class, etc.**

// boton creado dinamicamente con id = btnDinamico

**document.addEventListener( ‘click’, function(e) {**

**if ( e.target.id == ‘btnDinamico’ ) { console.log(‘soy el boton dinamico’) }**

**})**

**EVENTOS CON COMPORTAMIENTO AUTOMATICO.**

Los elementos con eventos autaticos predeterminados, tienen la particularidad de lanzar un evento preestablecido. Si un elemento con este tipo de comportamiento le asignamos un codigo de ejecucion, primero ejecutara el codigo, y luego lanzará su evento predeterminado y recargará la pagina. Es ejemplo se puede ver en elementos con **a, submit ( boton ), formularios y otros.**

**La ejecucion de eventos automaticos se anula con la propiedad e.preventDefaul**

**MANIPULACION DE DOM EN EVENTOS DE BOM.**

Mediante eventos de BOM podemos manipular elementos de DOM.

Ej **window.addEventListener( ‘resize’, function() {**

**parrafo.innerText = ‘El screenview de de’ + window.outerwidth + ‘por’ window.outerheigth })**

**FASES DE UN EVENTO**

**Bubbling:** Ejecuta el handler del target que dispara el evento, y luego intenta ejecutar todos los handler hacia los elementos padres hasta llega al Dom.

**Capturing:** Dispara los handlers del elemento raíz hasta llegar hasta el target, a la inversa del bubbling.

Por defecto los eventos se disparan en fase bubbling pero se puede cambiar usando el ultimo parametro ( opcional ) del addEventListener.

**document.addEventListener( ‘click’, function() { codigo } , true )**

**btnbutton.addEventListener( ‘click’, function() { codigo } , true )**

En este caso, si ejecutamos el handler del target btn, primero se dispararan todos los eventos desde el documento hasta llegar hasta el target.

**Cancelacion de las fases del evento.**

Para esto, debemos hacer uso del evento construido por el lenguaje y por ende debemos declarar una variable para llamarlo.

**btnbutton.addEventListener( ‘click’, function(e) {**

**console.log(e)**

**})** mediante la variable “**e”** capturamos el objeto evento que se disparo

**Event**

Los gestores de eventos pueden estar atados a varios elementos en el DOM. Cuando un  
evento ocurre, un objeto de evento es dinámicamente creado y pasado secuencialmente a  
los handlers autorizados para la gestión del evento. La interfaz Event del DOM es entonces accesible por la función de manejo, vía el objeto de evento puesto como el primer (y único) argumento.  
Los objetos Event cuentan con propiedades genéricas para todos los objetos del mismo tipo y también personalizadas por evento por lo que cada uno va a venir con información relativa a la acción que se ejecutó, por ejemplo un evento click nos puede traer las coordenadas del puntero cuando se hizo click mientras que un evento keyup nos puede traer la tecla que se apretó. Además todos comparten una propiedad target la cual nos muestra cuál fue el elemento que inició la cadena de eventos.

**Target**

Es una referencia al objeto que disparo el evento.

**Stop propagation**

Esta propiedad nos sirve para detener la propagacion de un evento desde el handler en que se ejecuta.

**btnbutton.addEventListener( ‘click’, function(e) {**

**e.stopPropagation()**

**})**

**prevenDefault**

Esta propiedad nos sirve para prevenir la funcion que tiene por defecto un elemento html. Como por ejemplo un elemento **“a”** que por default redirecciona a un vinculo.

**a.addEventListener( ‘click’, function(e) {**

**e.preventDefault()**

**})**

**Eventos Customizados**

Jscript nos permite crear eventos personalizados que podemos disparar en el momento que querramos. Para esto tenemos que:

- Crear un evento

- Asignarlo a un elemento

- Despachar el evento

**//Creo un evento llamado "look" que se dispara en bubbling y no se  
puede cancelar**  
var evt = new Event("look", {"bubbles":true, "cancelable":false});  
document.dispatchEvent(evt);  
*// event can be dispatched from any element, not only the document  
miDiv.dispatchEvent(evt);*

midiv.dispatchEvent(evt);

**Eventos Dinamicos**

Se utilizan para asignar eventos a elementos creados dinamicamente, elementos que jscript no analizao al cargar el documento porque no existian.

Para esto tenemos que acceder al lugar del codigo donde se crea el elemento dinamico, intersectando el target del elemento en el momento en que nuesto script se ejecuta.

**// html**  
<button> Dinamico </button>

**// app.js**

var btn = document.querySelector(‘button’)

btn.addEventListener(‘click’, function(){

var btnDinamico = documento.createElement(‘button’)

btnDinamico.id = “dinamico”

btnDinamico.innerText = “Boton Dinamico !”

btnDinamico.addEventListener(‘click’, function(){

console.log(‘Boton Dinamico’) })

document.body.appendChild(btnDinamico) })

**// otra opcion.js**

var btn = document.querySelector(‘button’)

btn.addEventListener(‘click’, function(){

var btnDinamico = documento.createElement(‘button’)

btnDinamico.id = “dinamico”

btnDinamico.innerText = “Boton Dinamico !”

document.body.appendChild(btnDinamico })

document.addEventListener(‘click, function(e){

if ( e.target.id == ‘Dinamico’) console.log( “Boton Dinamico” })

**Eventos de JavaScript 1.0:**

|  |  |  |
| --- | --- | --- |
| Evento | Descripción | Asociado a |
| Click | Pulsación sobre un enlace o un elemento de un form | Ratón |
| MouseOver | Colocación del ratón sobre un enlace | Ratón |
| Load | Carga de la página. | Página |
| UnLoad | Salida de la página. | Página |
| Focus | Seleción de un elemento de un form. | Teclado/Ratón |
| Blur | Des-seleción de un elemento de un form. | Teclado/Ratón |
| Change | Edición de un elemento de un elemento text, textarea o cambio de un select en un forma. | Teclado/Ratón |
| Select | Seleción del campo de entrada de un elemento de un form. | Teclado/Ratón |
| Submit | Envío de un form. | Teclado/Ratón |

* **Eventos de JavaScript 1.1:**

|  |  |  |
| --- | --- | --- |
| Evento | Descripción | Asociado a |
| Abort | Interrupción de la carga de una página o imagen | Página |
| Error | Error en la carga o ejecuciónde una página o imagen | Página |
| MouseOut | Salida del ratón de un área o enlace. | Ratón |
| Reset | Pulsació de un reset en un form. | Teclado/ratón |

* **Eventos de JavaScript 1.2:**

|  |  |  |
| --- | --- | --- |
| Evento | Descripción | Asociado a |
| DblClick | Pulsación de doble Click sobre un enlace o un elemento de un form | Ratón |
| DragDrop | Se produce cuando se deja caer algo sobre la ventana. | Ratón |
| MouseDown | Pulsación de un botón del Ratón sobre un documento, enlace o form. | Ratón |
| MouseMove | Movimiento del cursor del Ratón. | Ratón |
| MouseUp | Liberación de un botón del Rtón sobre un docuemnto, enlace o form. | Ratón |
| KeyDown | Pulsación de una tecla. | Teclado |
| KeyUp | Liberación de una tecla. | Teclado |
| KeyPress | Cobinación de KeyDown y KeyUp. | Teclado |
| Move | Movimiento de la ventana o un frame. | Ventana |
| Resize | Redimensionamiento de la ventana o un frame. | Ventana |

**FORMULARIOS Y VALIDACIONES**

**Evento submit**

Es un evento que se aplica a formularios y acepta d**ualmente eventos de mouse y de teclado**.

Let form = document.getElementById(‘fomr’)

form.addEventListener(‘click’, (e) => {

e.preventdefault }

**Atributos de validacion de elementos de HTML**

Vienen por defectos con los form de html y son atributos minimos de validacion que serán validados al momento de enviar el formulario.

**type:** Establece el tipo de datos ingresados, ej: text, number, pass, mail..

**required:** Es campo es obligatorio de completar

**min y max:** Establece un minimo y un maximo numérico

**minlength y maxlength**: Establece un minimo y un maximo de caracteres

**pattern:** Establece una validacion asociada a una expresion regular (regex)

**Validacion estandard de HTML – Api ETMLElement**

Mediante el METODO Element.checkValidity(), podemos solicitar mediante java Script conocer el estado de validacion de un elemento de html que tenga una condicion de validacion establecida, para elementos sueltos que no pasen por el proceso de validacoin de un formulario. O sea, elementos que no esten incluidos en un form.

Por ejemplo,

tengo un campo input con la propiedad **required**

<html>

<input type=”text” required>

</html>

Asigno un evento a un boton que pedirá el estado de alguna propiedad de validacion de un elemento de html, siempre que este tenga establecida alguna condicion de validacion html. Esto me devolvera **true** o **false** si se cumple o no la condicion de validacion del elemento.

Ejemplo para api Element - **HTMLImputElement**

button.addEventListener(‘click’, ()=> {

input.checkValidity } // devuelve TRUE o FALSE

**Validacion Customizada**

Validacion sin usar los atributos de validacion HTML

Para esto buscamos el **valor del imput** en el momento de envio del formulario ( el valor para cualquier campo de control en formularios tanto para input, select, textarea se extrae mediante la propiedad .value )

uso del METODO **Element**.**setCustomValidity.**

Esta propiedad recibe un string como parametro.

form.addEventListener(‘submit’, (e)=> {

e.preventdefault()

let valor = input.value

let lingitud = valor.length

if ( longitut > 3 ) {

// envio del form al servidor.. //

}

else { input.setCustomValidity(“el campor debe tener mas de 3 letras”)

i} // esta api trabaja de forma particular, la api verifica si el imput cumple el requisito y devuelve como vacio el string de setCustomValidity(). Si, la condicion no se cumple, se toma el string del setCustom para enviar el mensaje de error.

**Funciones utiles para validacion**

uso del METODO **Element**.**setCustomValidity.**

Esta propiedad recibe un string como parametro.

form.addEventListener(‘submit’, (e)=> {

e.preventdefault()

let valor = input.value

let valorConTrim = valor.trim() // la funcion trim limpia los espacio a ppio y final de un string

valor.includes(“@”) // se utiliza para en strings y array para saber si esta incluido el valor especificado

encodeURIComponent() // es es una funcion suelta, que recibe un string como parametro, y lo que hace es transformar caracteres potencialmente peligrosos como para que estos puedan ser manejados por un servidor. Util para prevenir ataques XSS, por ejemplo si nos ingresan código javaScrip que no deseamos desde un input.

**REGEX – Expresiones Regulares**

Una regex es una secuencia de caracteres que conforman un patron de busqueda dentro de un String. ( Son universales en todos los lenguajes de programacion )

Son un objeto de Js que usa la siguiente notación.

Let regexp = /a/ // Se escriben entre barras y dentro se coloca el patron de busqueda.

/ abcd / => busca el patron abcd todo junto

\w => incluye todas las letras (word) y numeros

\W => negacion de word

\d => digitos

\D => negacion de \d

\s => saltod de linea, espacios, etc

Caracteres de **cantidad: repeticiones**

a{2} => aa

abc{2} => abcc

(abc){2} => abcabc

\*N => Entre cero y muchas instancias de N == {0,}

+N => Entre 1 y muchas instancias de N == {1,}

{n,m} => n es minima cantidad y m maxima cantidad

Caracteres de **posicion: ubicación**

^ => el primer caracter de todo un string ( no palabra, seria todo el parrafo )

$ => el ultimo carácter de todo un string ( no palabra, seria todo el parrafo )

\b => el inicio o final de una palabra ( seria **\ba** al inicio o **a\b** al final )

**APLICACION DE UNA EXPRESION REGULAR**

form.addEventListener(‘submit’, (e)=> {

e.preventdefault()

let valor = input.value

let regexp = *^$\w{,5,0}*

*if ( regexp.test(valor) ) { console.log(‘valido’) } // la funcion test arroja un boleano*

*else {‘No valido’} // y se le pasa el input como parametro*

**ASINCRONIA**

**PROTOCOLO HTTP**

**El protocolo http, es el protocolo estandar de la web para transferencia de informacion.**

**Partes del protocolo http:**

**- body**

**- headers**

**- content-length**

**- content-type**

**Jscript es un lenguaje asincronico y no bloqueante**

Se envía una petincion y esta es devuelta al instante. Entonces la ejecucion del codigo sigue y el callback ( funcion que administra la respuesta ) se encola para ser ejecutada en algun momento en nuestra aplicación, siendo administrada por el event loop.

**API WEB – XMLHttpRequest – XHR – AJAX (asynchronous jscrip and XML )**

**XMLHttpRequest** es una api web diseñada para obeter informacion asincronica de una url. Permite la obtencion de recursos de cualquier tipo de datos y puede ser utlizada con varios protocolos de transferencia incluyendo Http.

Construccion de una request con **XHR**

**var xhr = new XMLHttpRequest( );**

//////// Construccion de request con compatibilidad para navegadores viejos ///////

var xhr ;  
if (window.XMLHttpRequest) { // Mozilla, Safari, ...  
xhr = new XMLHttpRequest();  
} else if (window.ActiveXObject) { // IE  
xhr = new ActiveXObject("Microsoft.XMLHTTP");  
}

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

La creacion de una reques XHR nos devuelve un objeto con propiedades y metodos que podran ser utilizados para crear una conexión asincronica a traves del protocolo http.

**PROPIEDADES Y METODOS DEL OBJETO XHR**

**- readyState (**muestra el estado en que esta nuestra solicitud)

- 0 UNSENT objeto inicializado

- 1 OPENED objeto configurado

- 2 HEADERS\_RECEIVED el obj se envio y el server devolvio headers

- 3 LOADING Descargando. La propiedad response\_text mantiene datos parciales.

- 4 DONE La peticion se completo, no necesariamente exitosa

**- open()**

El metodo .open() nos permite configurar una solicitud saliente o una ya inicializada. Recive 2 parametros obligatorios, el **metodo** y la **url ( strings )**

Los metodos pueden ser: **PUT, PATCH, DELETE, GET** y **POST.**

**XMLHttpRequest.open( ‘GET’, ‘url’ )**

**- readyStateChange**

Todo objeto que extienda de la reques, tiene el evento readyStateChange que se dispara cada vez que el estado de la peticion se actualiza. Con este evento podemos controlar los estados de la peticion.

var xhr = new XMLHttpRequest( );

**xhr.addEventListener ( ‘readyStateChange’, function () {**

**console.log(xhr.readyState)**

**});**

**xhr.open(‘GET’, ‘url.com’)**

**- headers**

Los headers HTTP son metdatos que son enviados entre cliente y servidor al iniciarse una solicitud.

**- abort()**

El metodo abort se utliliza para abortar una solicitud si algo no nos gusta, por ejemplo una respuesta de algun header. Esto reinicia la peticion XHR y resetea toda su configuracion.

**- load()**

El evento load se dispara cuando el readyState es igual a 4, o sea cuando se completo la solicitud, sea exitosa o no. Esto sirve para chequear ciertas informaciones necesarias al completarse la solicitud.

**xhr.addEventListener ( ‘load’, function () {**

**if (xhr.status == 200 ) { entonces ejecuta algo }**

**});**

**- response**

La propiedad response se va a visualizar cuando la request se haya completado y se haya descargado la informaicon solicitada. Puede estar en varios formatos dependiendo de como hayamos configurado la propiedad responseType. Json, document, DomString etc. Muestra el contenido del recurso que hayamos solicitado

**- responseType**

Es una propiedad de configuracion y podemos establecer el formato en que querramos la respuesta.

**responseUrl**

Indica la url donde hicimos la peticion del recurso.

**Timeout**

Nos permite configurar el tiempo maximo de espera de la respuesta el ms. Para esto direcamente escribimos directamente sobre la **variable timeout().**

**xhr.timeout = 1 // espera 1 milisegundo**

Se puede agregar un addEventListener con ‘timeout’ como parametro para que ejecute algo si se dispara el timeout

**xhr.addEventListener('timeout', ()=> {**

**console.log('el pedido se ha exedido de tiempo')**

**})**

**- send()**

Este metodo se utliza para enviar la solicitud una vez se haya configurado la misma. Puede enviar parametros como argumentos en su interior siempre y cuando hayamos utlizado el metodo POST. Siempre va escrito al final de la configuracion de la respuesta

**xhr.send ()**

**- getAllResponseHeaders()**

Este metodo nos extrae todas las cabeceras en formato string dentro de una variable que declaremos.

**- getlResponseHeader()**

Este metodo nos extrae una cabecera en particular si sablemos cual es y lo pasamos como argumento en formato string.

**Let header = xhr.getResponseHeader( ‘content-type’ )**

**XHR y el DOM**

Jscript nos permite hacer peticiones xhr e insertar contenido html directamente dentro del Dom, esto evita tener que recargar una pagina entera cuando solicitamos algun recurso.

Ejemplo:

let btn = document.querySelector('button')

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

let xhr2 = new XMLHttpRequest

xhr2.open('GET', 'plantilla.html')

xhr2.addEventListener('load', ()=> {

if (xhr2.status == 200 ) {

let plantilla = xhr2.response

console.log(plantilla)

let div = document.createElement('div')

div.innerHTML = plantilla

document.body.appendChild(div)

}

})

xhr2.send()

})

**SAP – SINGLE APLICATION PAGE**

Una SAP es una pagina creada en una unica aplicación, donte todos los archivos html, css y Js se cargan de una única vez. Así mismo los recursos necesarios se van cargando dinámicamente según se vayan necesitando mediante la interacción de los usuarios. Las páginas no se tienen que volver a cargar y tampoco se transfiere a otra página.

En usa SAP los usuario no van a poder navegar entre el historial de paginas, pero si podran accader a distintos estados almacenados en el front-end mediante la API WEB HISTORY.

**API WEB HISTORY**

Nos permite manipular la informacion del historial de sesión del cliente para la pagina o frame en que se encuentre navegando.

Su interfaz se encuentra como propiedad de window en **window.history**.

- STATE

Porpiedad que representa el estado de navegacion correspondiente al estado actual de la URL y posicion de sesion del historial del cliente de la pestaña actual. Puede ser cualquier tipo de dato.

**MODULO V**

**ASINCRONIA**

Manejo de la memoria en JS

Un programa divide la memoria asignada en 2 grandes partes.

- **Heap:** Memoria para las tareas inmediatas de carga inicial.

**- Call Stack:** Memoria para almacenar tareas a ejecutarse a destiempo de la ejecucion del script.

Un programa en JS atraviesa 2 fases

1- Se ejecuta el script de inicio a fin ocupando el heap. En esta carga inicial se pueden agregar funciones asincronicas al call-stack.

2- Estas funciones asincronicas pasan al queue (cola) de llamadas, para ser ejecutadas una a una.

**Heap**

La memoria dinamica que se almacena en el heap se usa para almacenar datos que se crean en la ejecucion de un programa (casi todo el programa). Como por ejemplo **variables y constantes, funciones globales, y objetos.**

**Call-Stack**

Es una estructura dinamica de datos que almacena informacion sobre funciones activas de un programa. Se almacena informacion sobre las funciones y subrutinas que se estan ejecutando, que estan pausadas, o deban ejecutarse en determinado momento. Su **principal funcion es la capacidad de ejecutar varias funciones en paralelo** e ir haciendo un seguimiento del estado de cada una.

En el call-stack solemos encontrar **Event listeners, y funciones con patron call-back**

**PROCESO ASINCRONICO**

Es todo proceso que se ejecuta desde el call-stack y no en el heap. Es un conjunto de tareas asincronicas.

Cuando se llama a una funcion asincronica, se registra en el call-stack una l**lamada pendiente a esa funcion**. Cuando el evento de la funcion se ejecuta todas las funciones asociadas pasan del call-stack al queue para ser ejecutadas y usaran los recursos del heap.

**API PROMISE**

Usamos el call-stack para procesos asincronicos, y estos se componen de tareas asincronicas. Estas tareas se desarrollan mediante la API PROMISE.

**Una promesa** es una tarea asincronica con 3 estados posibles: **Pendiente, resuelta, rechazada.**

\* Toda promesa se inicia como pendiente. Al ejecutar new Promise pasamos una funcion que se ejecutara autmaticamente.

\* Dentro de la funcion podemos definir el estado de la promesa como Resuelta o Rechazada.

\* Por fuera del constructor se definen callbacks, que seran registrados en el call-stack y se ejecutaran en el cambio de estado de la promesa.

**PROMESA**

- Es una tarea asincronica con 3 estados posibles, pendiente, resuelta o rechazada.

- Se crean con el constructor **new Promise.**

**FUNCIONAMIENTO DE API PROMISE**

1- La promesa empieza con el estado de pendiente. Al ejecutar new Promise Js lee en ese instante la funcion pasada al constructor y ejecuta linea por linea cada instrucción.

2- El ejecutor recibe 2 argumentos. El primeo es la funcion que al llamarse cambia el estado de la promesa a resuelta.

3- El segundo argumento es la funcion que cambia el estado a rechazada (rechazada no es necesariamente error).

Las promesas deben ser usadas para operaciones de call-stack.

Pueden haber promesas que no tengan el estado rechazado.

Registrar callbacks para ser ejecutados según la promesa cambie de estado.

**Promise devuelve un objeto con 3 metodos.**

**.then(function)** Ejecuta la funcion dada si la promesa se resuelve.

**.catch(function)** Ejecuta la funcion dada si la promesa se rechaza.

.**finally(function)** Ejecuta la funcion dada independientemente de su estado.

**Una vez que una promesa es resueta devuelve los metodos then y catch, y estos traen como callback el valor de la promesa resuelta. Este valor no lo podemos acceder como propiedad de una promesa porque viene como metadatos. La unica manera de acceder a estos es mediante el then y el catch a traves de un callback.**

Encademamiento de metodos de la promesa

Un promesa puede encadenar sus metodos de respuesta retornandolos dentro de los mismos. Por ejemplo, dentro del then de una promesa, puedo crear otra promesa y retornarla y encadenar otro then en respuesta a la nueva promesa.

**FUNCION ASINCRONICA**

Es una funcion comun declarada con la palabra **async** que permite el uso del operador **await** dentro de ella.

El opérador await espera a que una promesa se resuelva y retorna el valor pasado a la funcion resolver. En vez de callbacks se utliliza un simple operador.

- Se ejecuta la promesa dada

- Pausa la ejecucion de la funcion hasta que la promesa se resuelva.

- Al resolverse la promesa await retorna el valor pasado como argumento a la funcion resolver.

- En caso de rechazarse la promesa, await termina la ejecucion de la funcion y ejecuta el .catch asociado.

**ENCADENAMIENTO DE PROMESAS**

ARQUITECTURA DE PIPELINE

Consiste en ir transformando el flujo en un proceso de varias fases secuenciales, donde la salida de cada una es la entrada de la anterior.

Beneficios de Pipeline:

\* Programcacion mucho mas declarativa

\* Division de un proceso en operadores reutilizables

\* Codigo mas testeable al poder intercambiar operadores de un proceso.

Casos de uso:

\* Generacion de reportes complejos.

\* Procesamientos intercambiables

\* Division de la interfaz de un proceso.

Para implementar la arquitectura pipeline, tenemos 2 partes

1- Operadores o Stages,

2- La tuberia o pipeline,

**API FETCH**