**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.**

- Trabajan con un unico flujo de datos

- Se usan con una unica data asincronica de respuesta.

- No son simples de cancelar.

**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**

Es una interfaz para solicitar recursos de manera asincronica.

Es una API que unifica las Api:

XHR + Promisse + Response + Request + Body + Headers

ej: let mifetch = fetch(/url)

A direrencia de las promesas tradicionales, fetch crea una promesa y los valores de esta promesa son devueltos en un objeto tipo **response** y no en metadatos como las promesas tradicionales.

Este objeto tipo response trae valores de body, headers, status, type y mas.., a la vez que metodos includos dentro del mismo.

Como la variable fetch es una promesa, podemos encadenar respuestas usando los metodos then y catch.

Ej: mifetch.then (valor => {

console.log(valor)

})

Tener en cuenta que valor es un objeto response de la respuesta que solicitamos y no el valor de la respuesta y trae muchos datos y metodos dentro.

Ej: mifetch.then (valor => {

return valor.text()

})

Si usamos un metodo del objeto response para tratar de traer un valor de respuesta, se vuelve a generar una promesa con esta respuesta.

let mifetch = fetch(info.txt) // url

mifetch

.then(valor => { // valor = objeto response

return valor.text() // promise {}

})

.then(valor => {

console.log(valor) // String..

Resumiendo, cuando solicitamos un recurso asincronico con fetch, la funcion fetch recibe una url y devuelve una promesa que cuando se resuelve devuelve un objeto response. Este objeto response tiene metodos para convertir el contenido de la respuesta en formatos procesables, como .json(), .text() y otros. Cuando ejecutamos algun metodo del objeto resopnse nos vuelve a retornar una promesa, que cuando se resuelve nos devuelve la respuesta en el formato del metodo seleccionado. Por esto es comun encontrarnos con 2 then cuando usamos fetch. El primer then para darle formato a la respuesta de salida y el segundo para interceptar la respuesta transformada en valor.

**OPERACIONES**

**Biblioteca RxJS (Reactive extensions for js)**

La programacion reactiva es un paradigma de programacion asincronica que se ocupa de los flujos de datos y la propagacion del cambio.

RxJS es una biblioteca para programacion reactiva que utiliza **observables** que facilitan la composicion del codigo asincronico o basado en devolucion de llamada. Proporciona una implementacion del tipo observable y funciones para crear y trabajar con observables.

Estas funcinoes se suelen utilizar para:

\* Conversion de codigo existente para operaciones asincronicas en observables.

\* Iterar a traves de los valores de una secuencia.

\* Mapeo de valores a diferentes tipos.

\* Filtrado de flujos.

\* Componer multiples secuencias.

\* Funciones de creacion observables a partir de eventos, temporizadores y promesas.

**OBSERVABLES**

- Trabajan con un flujo continuo de datos ( escuchan una gran cantidad de respuestas )

- Si fallan puedes ejecutar comandos y reintentar continuar con el observer.

- Se pueden encadenar operadores adicionales como map, filter, reduce y muchos mas.

- Existen operadores mas potentes como retry() o replay()

- Pueden ser creados desde otras fuentes como eventos.

- Son funciones a las cuales podemos suscribirnos en multiples lugares.

**OPERADORES**

Son funciones que se basan en la base de observables para permitir una manipulacion sofisticada de las colecciones.

Ej: map(), filter(), concat() y flatmap()

**PIPES**

Se utiliza para vincular operadores entre si. La funcion pipe() toma como argumento las funcinoes que desea combinar y devuelve una nueva funcion que cuando se ejecuta, ejecuta las funciones compuestas en la secuencia.

Un conjunto de operadores aplicados a un observable es una receta, un conjunto de instrucciones para producir valores deseados. La receta x si misma no hace nada, debe llamar a suscribe() para producir algun resultado.

**MODULO VI**

**PARADIGMAS**

**PROGRAMCAION IMPERATIVA**

Es un paradigma de programacion que utiliza formulas que alteran el estado de un programa.

Se encadenan una serie de instrucciones que determinan lo que el ordenador debe hacer para alcanzar en cada momento un resultado determinado.

Los valores de las variables se modifican mientras se ejecuta el programa, de tal manera que para gestionar las instrucciones se integran estructuras de control como bucles o estructuras anidadas en el codigo fuente.

Un codigo imperativo **describe paso por paso el como.**

**PROGRAMACION DECLARATIVA**

Se basa en describir el problema declarando propiedades y reglas que deben cumplirse, en lugar de instrucciones.

Las solucinoes se obtienen mediante mecanismos internos de control, sin especificar como encontrarlas exactamente.

No existen asignacinoes destructivas y las variables son utilizadas con transparencia referencial.

Los lenguajes declarativos son razonados matematicamente.

Un codigo declarativo **describe el que y ocultando el como**.

**PROGRAMACION FUNCIONAL**

Es un tipo de programacion declarativa centrada en el uso de funciones para realizar operaciones.

Las funciones son datos que se pueden manejar, pasar, componer, obtener.

Es una **programacion declarativa implementada mediante funciones**.

Funcion:

Representa una secuencia de acciones que tienen un resultado o fin concreto y especifico.

Su uso basico es la transformacion de informacion, datos, procesos, subrutinas, etc.

\* **Entrada**: Son los valores de entrada o argumentos de una funcion. Sirven para parametrizar las operaciones de una funcion.

\* **Operacion:** Instruccion o accion dentro del proceso de una funcion. Una funcion es un conjunto de operaciones secuenciales.

\* **Salida o retorno:** Es el valor devuelto al programa principal. No debe confundirse con salida en pantalla o persistencia.

“Side effect” Es todo aquello que puede ocasionar que una funcion, dadas las mismas entradas, retorne diferentes salidas.

**PROTOTIPOS**

Son un conjunto de normas que permiten integrar la POO en Js.

Construccion de un prototipo ( 3 formas )

1- Funciones constructoras

2- Create

3- Literal

**Funciones Constructoras:**

function persona ( nombre, edad ) {

var p = {

nombre: nombre,

edad: edad

}

return p

}

var juan = persona(“Juan”, 20)

var maria = persona(“Maria”, 35)

Una funcion constructora retorna un objeto que al ser referenciado por una variable crea un nuevo objeto. Esto se debe a que la funcion crea una variable objeto dentro de su ambito privado y la retorna al llamador de la funcion, para luego una vez realizada la tarea de la funcion reiniciar su cuerpo y quedar disponible para ser llamada nuevamente con otro llamador.

**Literal**:

let a = {} // \_\_proto\_\_

Todas las propiedades dentro de un objeto creado literalmente son dinamicas, podemos modificarlas, borrarlas, iterarlas etc.

**Create**:

let prototipo = {

saludo: function() {

console.log(‘hola’)

}

}

Creamos un objeto que lo vamos a usar como prototipo para la creacion de un nuevo objeto b,

let b = Object.create(prototipo) // \_\_proto\_\_ = parametro

Al usar el metodo Object.create creamos un objeto totalemente nuevo y vacio que se aloja en una nueva variable, y su prototipo va a ser lo que se le pase como parametro. Teniendo en cuenta **las reglas de la cadena de prototipos, si un objeto no encuentra una propiedad dentro de si mismo, puede bajar tantos prototipos como encuentre su cadena hasta encontrarla y ejecutarla o arrojar error**.

En este ejemplo, b no contiene la propiedad saludo, pero puede ejecutarla ya que se encuentra en su cadena de prototipos.

**Configuracion de las propiedades de un objeto.**

Let a = Object.create(null, {

x: {

value: 1 // su valor real

writable: false // no admite escritura

enumerable: false // no se puede iterar

configurable: // no admite borrado

}

En este caso creamos un objeto con Object.create, y lo que hacemos en no pasarle un prototipo (null) y le pasamos una propiedad ( x ) con un objeto de configuracion como valor. Estas son las configuraciones de un objeto que permiten realizar ciertas acciones sobre el mismo. Todo objeto creado con Object.create, solo necesita un valor, y las 3 opciones restantes vienen por defecto en false.

**OPERADOR NEW EN JS**

El operador new seguido de la invocacion de una funcion se llama constructor y realiza las siguientes acciones internas:

1- Crea e inicializa un objeto nuevo completamente vacio

2- La redefine en el contexto con el objeto que acaba de crear

3- Retorna el objeto creado.

This seria el objeto nuevo que es automaticamente retornado.

Function Persona (nombre, edad) {

this.nombre = nombre

this.edad = edad

}

var Juan = new Persona(“Juan”, 20)

Juan es un nuevo objeto del tipo o clase “Persona”

*Juan*

*> Persona {nombre: “Juan”, edad: 20} //* Prefijo persona

**PATRONES: ESTRATEGICO Y TACTICO.**

Para analizar un paradigma utilizaremos 3 niveles.

\* **Estrategico:** Es el nivel de los principios de diseño, las directrices mas generales del programa.

\* **Nivel tactico:** Formas de organizar el codigo basandonos en los principios estrategicos.

**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.

cos.

\* **Nivel Operativo:** Es la implementacion tactica en un escenario concreto de dichas tacticas.

Desarrollo del **nivel estrategico** basado en los principios de Programcacion Funcional.

**- Funciones como valores.** Un principio fundamental es tratar a las funciones como valores. En PF una funcion es un tipo de dato mas. Se puede almacenar, crear de forma aislada, pasar como parametro, retornar, etc.

- **Funciones Puras.** Otro principio fundamental es dividir el codigo en funciones Puras. Son funciones que no tienen efectos secundarios, o sea, paa el mismo juego de entradas devuelve el mismo valor de salida.

- **Funciones de orden superior.** Es una funcion que sirve de algoritmo para crear otra funcion. Basicamente es una funcion que retorna otra funcion.

Nivel tactico.

Es el nivel de los patrones de diseño. Todos los paradigmas ofrecen estrategias para resolver problemas. Basicamente son 4 las mas comunes.

**\* Callback**

Es una funcion que se pasa como parametro a otra funcion, ej: .map, .filter, .find, .reduce, un eventlistener, una promesa.

**\* Recursividad**

Es una funcion que se invoca a si misma con diferentes valores

**\* Composicion de funciones**

Es el uso de Funciones de orden superior

**\* Pipeline**

Es una sucesion de funciones que se ejecutan una seguida de la otra. Lo mas importantes es que todas las funciones encadenadas tengan el mismo tipo de retorno y los mismos valores de entrada.

**FUNCIONES EN JS**

Como TODO en JS es tambien un objeto.

- Una funcion tambien se puede **manejar como un objeto**.

- Podemos **definirle una propiedad** en su interior y manejar sus valores, ej:

foo()

foo.x = true

- Las funcinones en Js son **variadicas**, o sea no dependen de la cantidad de argumentos definidos o de parametros pasados para que estas funcionen. Si la cantidad de argumentos no coincide con los parametros funciona igual.

- Tienen **ambito** (scope) y **contexto.**

**\* AMBITO:** Es el alcance que tiene una funcion para llegar a una variable determinada. Puede ser, externa o global, interna o local o variables preinicializadas o por parametro.

\* **CLOSURE:** Es el espacio que se genera entre una funcion que es definida adentro de otra funcion. Todas las variables que esten en ese espacion interno, van a formar parte del closure dela funcion interna.

Es la combinacion de un objeto de funcion y un alcance (un conjunto de enlaces de variables) en el que se resuelven las variables de la funcion.

Ej:

function externa(x) {

return function interna(y) {

console.log(y+x)

}

}

var resultado = externa(50)

La variable resultado es referencia de funcion interna, si ejecutamos por ejemplo interna(90), funcion interna tomara como parametro 90, pero usara el parametro x = 50, que vive en el closure de funcion interna.

Podemos ver el valor de closure alamcenado en la propiedad

funcion interna.scope

**\* CONTEXTO**: Es una referencia a la estructura del mismo objeto que contiene a esa funcion. Podemos acceder al contexto de una funcion usando la palabra reservada **this.** En Js el contexto de una funcion **no es estatico.** Este contexto cambiara según el metodo en que se llame a la funcion, normal, call, apply, bind. Basicamente el contexto es el lugar desde donde es llamada la funcion.

**Llamadas a funciones:**

**CALL:**

function foo(a, b) {

console.log( this a b )

}

**foo.call(objet, a, b)**

Cuando llamamos a un funcion con call, aparte de los parametros de la funcion podemos pasarle como primer parametro un objeto, al cual la funcion hace referencia (para usar con this) y luego los parametros normales establecidos en la funcion separados por coma.

**APPLY:**

Funciona exactamente igual que call, con la diferencia que los parametros normales de la funcion se pasan **dentro de un array** en lugar de pasarse sueltos y separados con comas.

**BIND:**

El metodo bind() crea una nueva funcion, que cuando es llamada, asigna a su operador this el valor entregado, con una secuencia de argumentos dados precediendo a cualquiera entregados cuando la funcion es llamada.

Foo.bind(this.argumento [Arg1, Arg2, ..])

this.Arg es un valor que sera entregado a la funcion de destino cuando se llame a la de enlace.

En terminos generales, la funcin bind() crea una nueva funcion (funcion ligada) con el mismo cuerpo que la funcion que sera llamada con la referencia this asociada al primer argumento de bind() el cual no podra ser sobreescrito.

this.x = 9;

var module = {

x: 81,

getX: function() { return this.x; }

};

module.getX(); // 81

var getX = module.getX;

getX(); // 9, porque en este caso, "this" apunta al objeto global

// Crear una nueva función con 'this' asociado al objeto original 'module'

var boundGetX = getX.bind(module);

boundGetX(); // 81

**MODULO VII**

**PROTOTIPOS**

JS es un lenguaje basado en prototipos. Respecto a la herencia, Js solo tiene una estructura de objetos. Cada objeto tiene un propiedad privada **([[Prototype]])** que mantinene un enlace a otro objeto llamado su prototipo.

El objeto prototipo tiene su propio prototipo y asi sucesivamente hasta que se alcanza un objeto cuyo prototipo es null. Null no tiene prototipo y actua como enlace final en la cadena de prototipos.

Casi **todos los objetos en Js son instancias de Object**, que se situa a la cabeza en la cadena de prototipos.

El objeto prototipo se utiliza como plantilla a partir del cual se obtiene el conjunto inicial de propiedades de un nuevo objeto.

Cualquier objeto puede especificar sus propias propiedades, tanto cuando se crea como tambien en tiempo de ejecucion. Ademas, cualquier objeto puede ser utilizado como el prototipo de otro objeto, permitiendo al segundo compartir las propiedades del mismo.

Cada objeto en Js tiene un segundo objeto Js asociado. Este segundo objeto se conoce como prototipo y el primer objeto hereda sus propiedades.

**3 FORMAS DE CREAR OBJETOS**

- Objetos **literales** : {}

- Operador new: **new Object()**

- Metodo create: **Object.create()**

**LITERALES**: Todos los objetos creados por literales tienen el mismo objeto prototipo (prototipo base para todos los objetos) y podemos referenciarlo mediante **Object.prototype.**

**FUNCIONES CONSTRUCTORAS**: **Un constructor es una funcion diseñada para la inicializacion de objetos creados recientemente**. Se invocan con la palabra **new** y automaticamente se crea un nuevo objeto, por lo que el constructor solo necesita inicializar el estado de ese nuevo objeto. Una caracteristica fundamental de los constructores es que su propiedad **prototupe** se usa como prototypo del nuevo objeto.

La creacion de un objeto comprende 3 fases, crear el objeto, establecer sus propiedades y retornarlo. Cuando usamos el operador **new** las fases de crear el objeto y retornarlo ya estan implicitas. Por lo tanto usamos el operador **this para apuntar al objeto donde se esta instanciando la nueva variable** y tener asi una referencia de cual sera el objeto al cual le asignamos las propiedades.

Las func. Constructoras son:

- Funciones de primer orden. // No tienen restricciones de uso.

- Son Variadicas. // Aridad indefinida. Acepta un numero variable de argumentos.

- Tienen Ambito // Es el alcance de la funcion

- Tienen contexto // Es el objeto al que pertencece la funcion cuando se ejecuta y se referencia por la palabra clave this.

**API OBJECT:** El metodo **object.create()** crea un nuevo objeto y utilza su primer argumento como prototipo de ese objeto. Tambien toma un segundo argumento que describe las propiedades del nuevo objeto.

**Object.create es una funcion estatica**, no un metodo invocado en objetos individuales. Para utilizarlo, se pasa el objeto prototipo deseado como argumento.

Var Persona = {

saludo: function() {

console.log(“Hola”)

}

}

var Juan = Object.create(Persona)

Esto nos provee la capacidad de crear un objeto con un prototipo arbitrario, en otras palabras, un “**heredero”**. Sin pasar por una funcion constructora.

**PROTOTIPOS EN FUNCIONES CONSTRUCTORAS**

functino persona () { }

var Juan = new Persona()

Cuando creamos un objeto a traves de una fucion constructora, este tendrá su propio prototipo de objeto, como asi tambien heredara el prototipo (objeto prototype) del constructor (funcon constructora) que sera el objeto **prototype** que se encuentra dentro de la funcion. Podemos entonces manipular los prototipos de la funcion constructora y estos tambien seran heredados por todos los objetos creado o instanciado por la funcion. De este modo si hacemos

\_\_proto\_\_ ( prototipo de Juan ) tendrá el contenido del propiedad:

prototype de el constructor.

Persona.prototype.class = Hombre

la variable Juan heredara la propiedad classe = Hombre en su prototipo. Asi podemos crear **pseudo “clases”** en una funcon constructora cuyo valor se transmitira al prototipo del new objeto creado.

Juan.\_\_proto\_\_.class = “mujer”

OJO ! Si cambiamos el valor de la propiedad class, desde una instancia en particular desde el objeto prototipo, este valor se vera reflejado en todas las instancias que salgan de la misma funcion, ya que el objeto prototipo pasa por **referencia.**

El formulado ideal seria que la funcion constructora tuviese las propiedades que son unicas para cada objeto, y en el prototipo de la funcion constructora declara las propiedades que son comunes a todos los objetos.

Todas la propiedades que se encuentren en el objeto prototype de la funcion constructora serán compartidas por todas las intancias de esta funcion, mientras que los valores asignados a la propiedad **this** de la funcion, sera valores unicos para cada instancia.

( Esto es asi, porque cuando usamos el operador new, internamente se crea un nuevo objeto, cuyos valores son asignados a traves de this y ese nuevo objeto es retornado por la funcion constructora

ej:

function Persona(nombre, edad) {

this.nombre = nombre

this.edad = edad

}

Persona.prototype.saludo = function () {

console.log(“Hola, buen dia”)

var Juan = new Persona(“Juan”, 35)

**MODELO PROTOTIPOS - HERENCIA CON OBJECT.CREATE**

En el modelo prototipos podemos crear prototipos que hereden de un prototipo superior, o sea unA subclase que hereda de una superclase.

Var persona = {

saludo: function() {

console.log(“Hola”)

}

}

var empleado = Object.create (persona, {

trabajar: {

value: function () {

console.log(“Trabajando”)

}

}

})

var Juan = Object.create(Persona)

En este MODELO, con object.create podemos crear una subclase, o subprototipo. La subclase empleado creeada con object.create, recibe como parametro el objeto que determina la clase persona, y luego los valores propios del objeto. Juan contines en su interior el prototipo de empleado y dentro de este el de persona.

Notese que en este ejemplo solo heredamos metodos.

Ejemplo 2:

function Persona(nombre, edad) {

this.nombre = nobre

this.edad = edad

}

Persona.prototype.saludo = function() {

console.log(“hola”)

} // añadimos un metodo a persona.

Function Empleado (sueldo) {

this.sueldo = sueldo

}

Empleado.prototype = Object.create(Persona.prototype) // Añadimos metodsos a Empleados a traves del prototype de persona.

Empleado.prototype.calcularSueldo = function () {

console.log(“Calculando…”) } // Añadimos nuevo metodo a Empl.

Var Juan = new Empleado (100) // creamos juan a pártir de Empl.

En este caso Juan solo recibe el valor de empleado mas los metodos de empleado y los que empleado hereda de persona. Si queremos que a su vez reciba las propiedades y valores de Persona tenemos que reescribir empleado de la siguiente manera.

Function Empleado (sueldo, nombre, edad) {

this.sueldo = sueldo

Persona.call(this,nombre,edad)

} // Aqui llamamos a **Pesona con call** para llamarla desde el **contexto** de empleado que seria usando el **primer parametro “this”** de Persona.call, los otro dos parametros son los valores originales que recibe la funcion. Y cuando creemos la instancia Juan, debera realizarse con todos los parametros.

Var Juan = new Empleado(100, “Juan”, 35)

**METODO DE CLASES**

ES6 permite la sintaxis **class**, para simular la creacion de clases

El constructor class tiene los mismos prototipos de una funcion. O sea, un definidor de clases es una funcino. Es una funcion especial cuya principal diferencia es el hoisting.

Declaracion de clase

Class Poligono {

constructor...

}

Expresion de clase:

var Poligono = class {

constructor…

}

La funcion class **no recibe parametros**, estos son colocados dentro de la funcion interna **“constructor”**que es la encargada de recibirlos desde la creacion de la variable. El constructor puede utilizar la palabra reservada **super** para llamar al constructor de una **superclase**.

class Personas {

cosntructor(nombre){

this.nombre = nombre

}

}

let persona = new Persona(“Maria”)

**DEFINICION DE METODOS EN UN PROTOTIPO TIPO CLASS**

Para definir un metodo en un prototipo class, simplemente declaramos el metodo dentro de class. De esta manera se sobreentiende que cuando hay un metodo suelto dentro de un constructor class, directamente ira a para al objeto prototype de las instancias. ( *maria.prototype.saludo = function () {console.log(“hola”)}* ).

Esta propiedad no admitira sobreescritura, ni borrado, ni iteraciones y sera compartido a traves de la cadena de prototipos para todas las instancias.

class Personas {

cosntructor(nombre){

this.nombre = nombre

}

saludo() {

console.log(“hola”)

}

static saludo2 {

console.log(“chau”)

}

}

let persona = new Persona(“Maria”)

Cuando añadimos la palabra **static** antes de la creacion del metodo, este ira a parar directamente como propiedad de la clase y no del prototipo de las instancias. No se podra acceder a ellos desde las instancias, y solo se podra acceder directamente desde la funcion class que lo contiene. Este metodo no sera commpartido a lo largo del prototipo.

***Las funciones tipo clase unifican la creacion de una instancia mediante funciones constructoras y mediante modelo de prototipos ya que incluyen los beneficios de ambas en una misma sintaxis:***

***- prototipos heredables (propiedad prototype)***

***- valores de instancia (a traves de la funcion constructora y this)***

**HERENCIA ENTRE CLASES – actualizado ESC6**

Para crear herencia entre clases y extender tanto metodos como propiedades usamos a palabra reservada **extends** para extender una clase a una clase superior, y la palabra **super** para invocar el constructor de la clase superior.

La funcion **super** de la subclase incluira ademas los parametros del constructor de la clase superior.

Class Persona {

constructor ( nombre, edad) {

this.nombre = nombre

this.edad = edad

}

saludo() {

console.log(“hola…”)

}

}

Class Empleado **extends** Persona {

constructor (sueldo, nombre, edad) {

**super**(nombre, edad)

this.sueldo = sueldo

}

trabajar() {

console.log(“trabajando…”)

}

}

let empleado = new Empleado(100, “Juan”, 35)

**GETTERS Y SETTERS EN CLASES**

Los metodos get y set se utilizan para acceder y modificar atributos de un objeto. El nombre de los getters y setters no puede ser el mismo que la propiedad porque se produciria un bucle.

Class Telefono {

constructor(marca) {

this.\_marca = marca

}

get marca() {

return this.\_marca

}

set marca(mar) {

this.\_marca = mar

}

}

let miTelefono = new Telefono (“Samsung”)

Creamos un objeto telefono con propiedad marca = “Samsung”

miTelefono.\_marca = “Nokia”

Seteamos el valor de la propiedad marca a “Nokia” ( SET )

mensaje.innerHtml = “mi telefono es un” + miTelefono.marca

Mostramos el valor de la propiedad marca de miTelefono ( GET )

**MODO IFEE – Inmediatly Invoked Function Expresion**

Se trabaja todo el programa dentro del cuerpo de una funcion para proteger la privaciad de las variables.

(Function () {

cuerpo del programa

}) ()

para ejecutarla ponemos toda la funcion **anonima** dentro de parentesis y seguido al cierre colocamos () para ejecutarla.

**MODO ESTRICTO**

<script>

“use strict”

</script>

Activamos modo estricto declarando la frase “use strict” en **la primer linea del script.**

**MODULO VIII**

**PATRONES E ITERADORES**

**PROXY**

Permite crear un intermediario para otro objeto, que puede interceptar y redefinir operaciones fundamentales para dicho objeto.

Se crea con **new Proxy** y recibe 2 parametros

\* **Target**: objeto objetivo

\* **handler**: objeto que define las operaciones interceptadas y las redefiniciones.

El constructor new Proxy retorna un objeto con las mismas propiedades y valores que el target. Esta pensado para utlizarse en patrones proxy, como un vigilante de la comunicación entre objetos.

En el **handler** podemos interceptar lecturas (**get**) y escrituras (**set**) de las propiedades de un objeto.

**PATRON MODULO**

Es el uso o implementacion constante de funciones anonimas autoinvocadas con todo nuestro programa adentro. Esto restringe nuestro programa y protege la integridad de nuestras variables.

Se separa la logica del programa en distintos archivos.js distintos ( modulos ) y dentro de cada modulo se crea una funcion autoinvocada contenedora.

PRINCIPÍO DE ENCAPSULAMIENTO

En cada modulo las variables **quedan** **encapsuladas** y mediante un espacio asignado en el objeto windows elegimos que variables queremos exponer al exterior.

< modulo\_1.js >

(function() {

var x = 1

function changeX (val) {

var x = val

console.log(x)

}

window.modulo1 = changex

})

De esta manera mediante modulo1() podemos acceder a la funcion changex desde el exterior.

**PATRON REVELADOR.**

Definimos una variable global pero su valor es una funcion autoinvocada. Dentro de la funcion colocamos nuestras variables privadas y retornos de funcion.

Var modulo1 = (function() {

var x = 10

**return** console.log(x)

}) ()

De esta manera a traves de la variable global accedemos al retorno de la funcion pero las variable locales sigen estando restringidas.

**PATRON SINGLETON**

Mediante este patron se busca que solo se pueda ***instanciar una unica vez*** la clase que definamos bajo este patron, y cada vez que realizamos un new, siempre recibamos la misma primer instancia.

**PATRON DE PUBLICACION Y SUSCRIPCION**

Se utiliza para desacoplar programas e individualizar las funcionalidades.

**ITERADORES**

Es un objeto que permite recorrer una colección y devolver un valor al finalizar. Implementa el protocolo de iteracion a traves del metodo **next()** que devuelve un objeto con 2 propiedades:

**\* Value:** Es el siguiente valor en la secuencia de iteracion.

**\* Done:** Es true si el ultimo valor de la secuencia ha sido consumido. Si value esta presente junto con done, es el valor de retorno del iterador.

Un iterador se considera finalizado cuando la invocacion de next() devuelve un objeto donde la propiedad done es true.

Una vez **creado**, el iterador puede utilizarse llamando repetidamente al metodo next().

Una vez **inicializado**, se puede invocar el metodo next() para acceder a las parejas clave-valor.

ITERABLES: Un objeto es iterable si implementa el metodo @@iterator. O sea que debe incluir, en si mismo o su cadena de prototipos, la propiedad **Symbol.iterator.**

**GENERADORES**

Los Generadores permiten definir un algoritmo iterativo al escribir una sola funcion que puede mantener su propio estado. Se crea con **GeneratorFunction** y funciona como una fabrica de iteradores. Cuando se ejecuta regresa un nuevo objeto Generador.

**STORAGE API**

Proporciona acceso al **almacenamiento de la sesion** o al **almacenamiento local para un dominio particular,**

**Window.sessionStorage**. Metodo para manipular el almacenamiento de sesion para un dominio.

**Window.localStorage**. Metodo para manipular el almacenamiento local de un dominio.

**Window.sessionStorage**

Permite acceder a un objeto de almacenamiento de sesion para el origen actual.

Estos datos almacenados en sesion storage, se borran cuando finaliza la sesion de la pagina.

La sesion de una pagina dura mientras el navegador este abierto y sobrevive a las recargas y restauraciones de la pagina. Abrir una nueva pagina o una nueva ventana provoca un nuevo incio de sesion.

**Window.localStorage**

Es una propiedad de solo lectura, y permite acceder a un objeto que guarda informacion en la memoria del navegador. Los datos almacenados se guardan en las sesiones.

Los objetos almacenados son clave-valor y se mantienen a traves de las cargas de la pagina. Se puede acceder a estos valores **como un objeto** o con los metodos **storage.getItem()** y **storage.setItem().**

Ej:

localStorage.colorSetting = '#a4509b';

localStorage['colorSetting'] = '#a4509b';

localStorage.setItem('colorSetting', '#a4509b');

**window.localStorage.setItem:** El metodo setItem() para un nombre y valor, y agregara esta clave al almacenamiento o actualizara el valor de esta clave si ya existe. Puede arrojar una excepcion si el almacenamiento esta lleno.

localStorage.setItem('colorSetting', '#a4509b');

**window.localStorage.getItem:** Este metodo devolvera el valor de una clave cuando se pasa el nombre de una.

Var value = storage.getItem(keyName)

**PERSISTENCIA DE DATOS EN EL NAVEGADOR**

Opciones para guardar datos en el navegador. Se accdede a estos a traves de la pestaña aplication del navegador.

**STORAGE – API WebStorage:**

Esta API almacena los datos en el objeto global windows y su objetivo reemplazar el uso de cookies. Solo **almacena strings.**

Alamcena los datos en arrays tipo [“indices” => valor] y para traer estos datos del almacenamiento se accede a los indices para traer el valor.

El tiempo de almacenamiento no se puede configurar y su espacio es limitado ( entre 5 – 10 mb ) y se divide en 2 tipos de almacenamiento:

**\* Persistente:**

localStorage.setItem(“usuario”,”Juan”) // Para almacenar valor

Este metodo solo alacena strings. Cuando intentamos almacenar numeros no hay problema porque automaticamente los convierte en strings, pero con los objetos no es igual.

Para almacenar objetos en localStorage usamos el metodo JSON.stringify para convertir un objeto de Json a string.

localStorage.setItem(“objeto”, JSON.stringify({x:1}))

Entonces se almacena el objeto en modo string. Nota: si el nombre de la clave se repite pisa el valor de la clave existente.

LocalStorage.getItem(“usuario”) // Accedemos al valor del dato solo con la clave.

LocalStorage.removeItem(“usuario”) // Borra una clave particular

LocalStorage.clear() // Limpia el almacenamiento.

**\* Temporal:**

sessionStorage.setItem

sessionStorage.getItem

sessionStorage.clear()

sessionStorage.removeItem(“usuario”)

Trabaja exactamente igual que local storage pero solo dura mientras la sesion este activa, o sea que el dominio este abierto. Cuando cerramos la navegacion los datos se borran automaticamente.

**Eventos Almacenamiento de la API STORAGE**

La API storage cuenta con un unico evento, **storage()** el cual se va a disparar (ejecutando su callback) cada vez que una particion del almacenamiento de la **API storage se modifique.** Esto va a pasar siempre y cuando estemos trabajando en pestañas distintas. El evento no es del DOM, por lo tanto siemrpe se asocia al objeto window.

Window.addeventListener (“storage”, e=> {

funcion….

}

)

**COOKIES**

**COOKIES:** Son archivos contenedores de informacion que se guardan dentro del navegador y persisten de una manera donde se puede configurar el tiempo de su existencia. A diferencia de Storage, las cookis son transmitidas en los protocolos http cada vez que se realiza un pedido externo

Las cookies **no disponen de una API**, por lo tanto no disponemos de metodos de control sobre ellas. Se accede a ellas a traves de **document.cookie** y se crean y configuran de manera manual. Sin embargo, sus propiedades son ampliamente configurables. (path, expires time, size, http, name, value, etc)

path // Lugar donde se guarda la cookie

domain // Dominio para el cual estara disponible

max-age // En segundos, determina la maxima duracion de una.

expires // En formato fecha determina la fecha de expiracion

secure // True-False, solo se transmite a traves de https (conexiones encriptadas.)

samesite // Determina si se envia a traves de distintos dominios.

Cada parametro de configuracion debera escribirse dentro del cuerpo de la cookie.

document.cookie = “**user**=Juan;path=/;domain=localhost; **user2**=..”

Solo **almacenan** **strings** y son datos asociados a valores. El problema de manejo de cookies en js es que para acceder a un valor tenemos que acceder a todo el contenido de document.cookie, ya que es un solo valor string que contiene todas las configuraciones.

Por esto se recomienda en JS tratar de usar local y sesion storage en lugar de cookies.

**API IndexedDB**

Es otro de los metodos de almacenaje de datos que nos permite tener una base de datos relacional en el navegador.

**MEDIOCION DE PERFORMANCE**

En Js disponemos de varia herramientas a la hora de medir la performance de nuestras aplicaciones y nuestro navegador.

**DESDE js**

**window.performance:**

El objeto global performance, nos muestra los tiempos **exactos** de la duracion de todas las operancines, como tiempos de carga, inicio, etc y se va a completar de forma asincronica por lo que las referencias de tiempos de ejecucion son sumamente utiles y referencian bien nuestros requerimietos.

Tambien podemos usar los metodos **date(), time(), timeEnd()** para calcular la duracion de nuestras operaciones en mls. Pero estos metodos no son los mas convenientes para medir performance.

**DESDE EL NAVEGADOR:**

**Pestaña Performance:**

Desde la pestaña performance del navegador, podemos hacer un **record** de las funciones que estamos ejecutando (tiempo, tipo, duracon, etc). Tambien es posible compartir el contenido de performance con otras personas.

**MODULO IX**

**NODE.JS**

**npm (node package manager – librerias, paquetes, frameworks etc.)**

Es el administrador de paquetes (bloques de construccion de codigos) mas grande del mundo. Consta de 3 componentes:

\* El sitio web

\* Interfaz de lineas de comando (CLI)

\* El registro

Se utiliza para:

\* Adaptar paquetes de codigo a aplicaciones o incorporarlos tal cual estan.

\* Descargar herramientas independientes.

\* Ejecutar paquetes sin descargar

\* Compartir codigo con cualquier usuario de npm

\* Restrigir codigo a desarrolladores especificos.

\* Formar ORG para corrdinar el mantenimiento, codificacion y desarrolladores de paquetes.

\* Formar equipos virtuales usando Orgs.

\* Administrar versiones y dependencias del codigo.

\* Actualizar las aplicaciones subyacentes.

\* Descubrir mutltiples formas de resolver el mismo rompecabezas

\* Encontrar otros desarrolladores que trabajan en problemas y proyectos similares.

**Node.js**

Es un entorno de ejecucion para correr Java Script construido con el motor V8 de Chrome.

Utiliza un modelo de operaciones E/S sin bloqueo y orientado a eventos. Esto lo hace liviano y confiable. Node.js incluye nmp en su intalacion.

Se dice que Node.js es Js del lado del servidor, o sea ejecutado en el Sistema Operativo.

**Un modulo Node.js** es basicamente nuestro archivo de desarrollo en Js, ya que node a medida que va leyendo nuestros archivos implementa un patron modulo encapsulando todo nuestro contenido y nuestras variables.

**Un modulo nativo de Node.js** son los modulos que vienen pre-cargados en node y nos permiten acceder a diferentes funcionalidades en nuestro Sistema Operativo. Son los que serian las API web para los navegadores. ( Ej modulo HTTP )

**Npm y Node.js**

Para administrar los paquetes npm instalados localmente, se recomineda crear un archivo **package.json** que:

\* Enumera los paquetes de los que depende el proyecto.

\* Permite especificar las versiones de un paquete que el proyecto puede usar.

\* Hace que la compilacion sea reproducible, por ende mas facil de compartir con otros desarrolladores. ( Publicarlo en npm )

Un paquete debe tener:

**“name”** // Lowecase, una palabra, sin espacios,– y \_ permitidos

**“version”** // Formato x.x.x, seguida de la especificacion del server

**Creacion de un package.json:**

> npm init

Esto iniciara un cuestionario en consola para ir configurando, y finalizara con la creacion de un package.json en el directorio en que se ejecuto el comando.

Para instalar un paquete usamos

> npm install –save Jquery // Esto instalara Jquery en nuestro archivo de configuracion como una dependecia real del proyecto.

**Instalacion de dependencias:**

Hay 2 formas, **locales** y **globales** y se elige en funcion de como se desea utilizar el paquete.

**Locales**: Son modules unicamentes disponibles en la carpeta raiz del entorno de desarrollo utilizado.

> npm itall <package\_name>

Esto crea el directorio **node\_modules** en el directorio actual y descarga el paquete a este directorio.

Globales: Se descargan las dependencias (paquetes) usando

> npm install -g <package>

**COMPILADORES Y TRANSPILADORES**

**TRANSPILADORES**

Son programas capaces de traducir el codigo de un lenguaje hacia otro o de una version del lenguaje hacia otra. Se utilizan en la etapa de desarrollo. Por ejemplo de ES6 a ES5 para que sea compatible con cualquier navegador.

**BABEL**

Es un **transpilador** Js que se encarga de compilar codigo Js para que este sea compatible en todos los navegadores independientemente su version. Lo que hace es traducir codigo Js moderno en codigo mas compatible con todos los navegadores.

Funciona mediante plugins que sirven para indicar que cosas queremos transformar.

Ej:

babel-plugin-transform-es2015-arrow-functions

babel-plugin-transform-react-jsx

Instalacion

> npm install –save babel-loader- bebel-core bebel-preset-env

babel-core: Nucleo o librería de Babel

babel-loader: Libreria para interactuar con Webpack

babel-preset: Para transpilar Js a codigo compatible.

**WEBPACK**

Es un **empaquetador de modulo (bundler)** que sirve para:

\* gestionar dependencias

\* Ejecutar tareas

\* Convertir formatos

\* Servidor de desarrollo

\* Carga y uso de modulos de todo tipo (AMD, CommonJs o ES2015)

Es muy util para aplicaciones web diseñadas con **filosofia** **modular**, es decir, el codigo soportado en modulos que luego se utilizan como dependencias en otros modulos.

Tambien puede usarse para **concatenacion de codigo, minimizacion y ofuscacion, verificacion de buenas practicas (linting), carga bajo demanda de modulos,etc.**

Algo muy interesante de Webpack es que no solo el codigo Js se considera un modulo, sino tambien **hoja** **de** **estilo**, **paginas** **html** e incluso **imagenes**.

Webpack es entonces una herramienta de compilacion (build tool) que coloca en un grafo de dependencias a todos los elementos que forman parte del proyecto,, Js, HTML, CSS, plantillas, imagenes, fuentes, etc.

De todos los archivos de codigo que presenta nuestro proyecto. Wbpack unifica y minimifica nuestros archivos en un unico archivo fuente de cada extension para que cuando un usuario intente acceder a nuestra pagina, tenga que hacer un unico pedido de recursos y no varios.

Instalacion

> npm install –save webpack-cli webpack-dev-server

Las librerias de babel-loader y babel-core son las necesarias para que Webpack pueda **integrar** la **transpilacion** del codigo.

Las librerias babel-preset son para que babel sepa como **interpretar** y compilar el codigo **JSX** en Js plano.

El paso siguiente es crear un archivo de configuracion para cada paquete, que se deben ubicar en el **directorio** **raiz** del proyecto.

1 - **.babelrc**

**{**

**“presets: [ @babel/env” ]**

**}**

2 - **webpack.config.js** (contiene entre otras configuraciones)

CONFIGURANDO WEBPACK.CONFIG

Como este archivo sera leido por NodeJs, debemos exportar el contenido de este archivo en una variable que pueda ser exportada desde un modulo node.

Module.exports = {

entry: “./app.js”,

output: {

filename: “bundle.js”,

path: \_dirname *// variable de node que hace referencia al directorio donde estoy parado*

}

module: {

rules: [ *// reglas para que webpack compile archivos Js. Excluyendo node\_modules y usando babel con los parametros que establescamos en el archivo* ***.babelrc.***

{

test: *\.js$*,

exclude: /node\_modules/

use: “babel-loader

}

]

}

mode: ….

}

- Entry: Archivo de entrada el programa para construir el bundle

- Output: Directorio y nombre de salida del bundle.

- Mode: Como Webpack va a compilar nuestros archivos.

- Module: Confiruraciones que aplican a cada modulo. Como Babel y Webpack van a compilar cada archivo.

**Recordar incluir en el package.json el script**

“scripts”: {

“dev”: “webpack”

}

Esto es para que podamos ejecutar > npm run dev

Entonces:

- Npm se comunica con webpack

- Webpack busca los archivos js a transpilar

- A medida que webpack encutra archivos js sabe que tiene que usar babel-loader y el metodo de este esta en .babelrc.

- Crea el archivo de salida minificado y listo para produccion.

**JSON - SERVER**

El paquete o modulo JSON-Server sirve para tener una API de REST completa y falsa con cero codificacion. Es un back end rapido para comenzar a prototipar front end.

> npm intall g json-server

Crear archivo json a servir ( **ej db.json** )

{

“posts”: [ { “id”:1, “autor”: Gustavo } ]

}

Iniciar servidor

> json-server –watch db.json

Consultar los datos

Si vamos a http//:localhost:3000/posts/1, tenemos

{ “id”:1, “autor”: Gustavo }

**CONSIDERACIONES:**

- Si realizamos solicitudes, POST, PATCH, PUT o DELETE, los cambios seran guardados automaticamente en db.json al usar **lowdb.**

- El json de su cuerpo de solicitud debe estar cerrado por el objeto, al igual que la salida GET.

Ej: { “nombre”: “Foobar” }

- Los valores del ID no son mutables. Cualquier valor de identificacion en el cuerpo de su solicitud PUT o PATCH sera ignorado. Solo se respetara un valor establecido en una solicitud POST, pero si no se ha realizado.

- Una solicitud POST, PUT o PATCH debe incluir un encabezado **application/type: application/json** para usar el JSON en el cuerpo de la solicitud. De lo contrario dara como resultado un 200 ok, pero sin cambios en los datos.

**TYPESCRIPT**

Typescript es un superconjunto de Js. Es un lenguaje construido sobre Js con añadido de caracteristicas tipicas de lenguajes fuertemente tipados. Typescript no sera compilado al final, sino que se transpilara a js puro.

**COMPROBACION DE TIPOS PRIMITIVOS**

La caracteristica principal de Ts es la **comprobacion de tipos estaticos.**

El type Check toma el tipo de dato de una variable, que es una palabra reservada seguida por “:” luego del nombre de la variable.

**const numero₁ : number = 10;**

**const texto : string = “hola”**

**const booleano : boolean = true**

**COMPROBACION DE CABECERAS DE FUNCION**

Con la misma logica que la comprobacion de tipos, podemos tambien indicar a Type Check que lance un error si una funcion no cumple con una **firma** **determinada**.

**Const sumar: (n1: number, n2: number) => number = function (n1,n2) {**

**return n1 + n2**

**}**

La **firma** de la funcion es el juego de parametros que recibe y el tipo de **datos** **que** **retorna**.

**( param: tipo, param2: tipo ) => tipo\_retorno**

Si la funcion **no** **retorna** **nada** podemos usar el tipo especial **void**.

**( param: tipo, param2: tipo ) => void**

**COMPROBACION DE PROPIEDADES DE OBJETOS**

Los **objetos** **se** **validan** por el type Check mediante una estructura propia denominada **interfaz**.

Cuando un objeto usa una interfaz, esta **obligado** a **respetar** la **estructura** de la misma. ( tener todas las propiedades obligatorias y respetar los nombres y tipos de datos definidos ).

Durante el desarrollo, esto permite hacer predecible la estructura de un objeto y darlo a conocer por toda la aplicación.

**TIPOS GENERICOS**

Permiten que el type check sea un poco mas flexible. Una declaracion de tipo generico actua como un placeholder de tipos de datos. El generico tiene un tipo de dato que la estructura desconoce y se determinara según su uso. Para usar un generico debemos encerrar su definicion entre los simbolos < y >. El contenido que este ahi dentro sera un identificador.