Sumario

Front-end....2
- Introducción....2
- Qué es y para qué nos sirve Javascript?....2
- ECMAScript....2

Sintaxis del lenguaje JavaScript....2
- Tipos de Datos....2
  - Primitivos....2
- Constructores....3
  - VAR....3
  - LET....3
  - CONST....4
- Notación de Punto Vs Corchete....4
  - Notación de corchete....4
  - Notación de punto....5
- Iteraciones....5
- Funciones Flecha....5

BOM y DOM....6
- BOM Browser Object Model....6
- DOM Document Object Model....7
  - Crear un nodo....7
  - Editar un nodo....7
  - Agregar / Remover Nodos del DOM....8
- Eventos....9
  - Introducción....9
  - Arquitectura de un evento....9
  - Definiendo Eventos....10
  - Element.addEventListener....11
  - Event....11
  - Target....11
  - stopPropagation....12
  - preventDefault....12
  - Eventos Customizados....13
  - Eventos de elementos dinámicos....13

AJAX....14
- Que es AJAX?....14
- Tecnologías incluidas en Ajax....14
- Ejemplo de Uso....15

Front-end

Introducción

Front-end y back-end son términos que se refieren a la separación de intereses entre una capa de presentación y una capa de acceso a datos, respectivamente.
Pueden traducirse al español el primero como interfaz, frontal final o frontal y el segundo como motor, dorsal final o zaga, aunque es común dejar estos por separado.
En diseño de software el front-end es la parte del software que interactúa con los usuarios y el back-end es la parte que procesa la entrada desde el front-end. La separación del sistema en front-ends y back-ends es un tipo de abstracción que ayuda a mantener las diferentes partes del sistema separadas. La idea general es que el front-end sea el responsable de recolectar los datos de entrada del usuario, que pueden ser de muchas y variadas formas, y los transforma ajustándolos a las especificaciones que demanda el back-end para poder procesarlos, devolviendo generalmente una respuesta que el front-end recibe y expone al usuario de una forma entendible para este. La conexión del front-end y el back-end es un tipo de interfaz.

Qué es y para qué nos sirve Javascript?

JavaScript (JS) es un lenguaje ligero e interpretado, orientado a 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.

ECMAScript

ECMAScript es una especificación de lenguaje de programación publicada por ECMA International. El desarrollo empezó en 1996 y estuvo basado en el popular lenguaje JavaScript propuesto como estándar por Netscape CommunicationsCorporation. ECMAScript define un lenguaje de tipos dinámicos ligeramente inspirado en Java y otros lenguajes del estilo de C. Soporta algunas características de la programación orientada a objetos mediante objetos basados en prototipos y pseudoclases. Desde el lanzamiento en junio de 1997 del estándar ECMAScript 1, han existido las versiones 2, 3 y 5, que es la más usada actualmente (la 4 se abandonó).
En junio de 2015 se cerró y publicó la versión ECMAScript 6.

Sintaxis del lenguaje JavaScript

Tipos de Datos

El último estándar ECMAScript define siete tipos de datos :

Primitivos

● Boolean: true y false.
● null : Una palabra clave especial que denota un valor null. Como JavaScript es case-sensitive, null no es lo mismo que null, NULL, o cualquier otra variante.
● Undefined: Una propiedad de alto nivel cuyo valor no es definido.
● Number: 42 o 3.14159.

● String: "Hola" , ‘hola’, `hola`
● Symbol (nuevo en ECMAScript 6).
● Object

○ Object : { indice : valor } Son matrices ordenadas de manera asociativa de un valor con un índice.
○ Array : [valor] Son matrices ordenadas de manera secuencial numérica ascendente.
○ Function : Son objetos con la habilidad de poder ser ejecutados.

Constructores

Para construir variables lo tradicional es que usemos el constructor ‘var’. A partir de ECMAScript 6 aparecieron dos nuevas formas de declarar una variable : let y const

VAR

Las variables inicializadas con el constructor ‘var’ cumplen con los siguientes tres puntos :

● Admiten redeclaración : Es decir que puedo volver a declarar una variable que ya había sido inicializada:

```
var a = true
var a = false
```

● Admiten redefinición : Es decir que puedo cambiarle su valor en cualquier momento del programa :

```
var a = true
a = false
```

● Admite alcance global : Es decir que salvo por los bloques funcionales, una variable declarada siempre es global, por lo tanto es una propiedad del objeto global window.

```
if (true) {
      var a = true
}
console.log(a)  //true
```

LET

Las variables inicializadas con el constructor ‘let’ cumplen con los siguientes tres puntos :

● No admiten redeclaración : Es decir que no puedo volver a declarar una variable que ya había sido inicializada:

```
let a = true
let a = false     //Error!!!
```

● Admiten redefinición : Es decir que puedo cambiarle su valor en cualquier momento del programa:

```
let a =  true
a = false
```

● No admiten alcance global : Es decir las variables declaradas pertenecen inicamente al bloque donde se crearon. Por otro lado, si las mismas fueran globales, lo serán pero no como propiedad de window.

```
if(true) {
      let a = true
}
console.log(a)          //Error!!!
```

CONST

Las variables inicializadas con el constructor ‘const’ cumplen con los siguientes tres puntos :

● No admiten redeclaración : Es decir que no puedo volver a declarar una variable que ya había sido inicializada:

```
const a = true
const a = false         //Error!!!
```

● No admiten redefinición : Es decir que no puedo cambiarle su valor

```
var a = true
a = false               //Error!!!
```

● No admiten alcance global :Es decir las variables declaradas pertenecen únicamente al bloque donde se crearon.

```
if(true) {
      const a = true
}
console.log(a)          //Error!!!
```

● Admiten redefinición de componentes internos

```
const obj = {
      x  :  1,
      y  :  2
}
obj.x = 10  // {x:10,y:2}
```

Notación de Punto Vs Corchete

En Javascript contamos con dos tipos de notación para poder ingresar a los datos internos de una matriz cualquiera. A diferencia de otros lenguajes en donde se suelen tener notación de sintaxis específicas para cada tipo de dato, acá ambas son para todos los tipos de objetos con la diferencia en que una nos permite acceder a más cosas que la otra. Observemos el ejemplo desde el siguiente objeto :

```
      const obj = { x  :  1,  y :  2, 0 : true}
```

Notación de corchete

● Índice Number : Puedo ingresar a un elemento usando obj[0]

```
obj[0]       //true
```

● Indice String : Puedo ingresar a un elemento usando obj[“x”]

```
obj["x"] //1
```

● Índice variable : Puedo ingresar a un elemento usando una variable
const indice = "x"
obj[indice]   //1

Notación de punto

● Índice Number : No puedo ingresar a un elemento usando un numero.

```
obj.0 //Error : Unexpected Number
```

● Índice String : Puedo ingresar a un elemento usando obj.x

```
obj.x       // 1
```

● Índice variable : No puedo ingresar a un elemento usando una variable como índice.

```
const indice = "x"
obj.indice //undefined
```

Iteraciones

Además de contar con las tradicionales iteraciones de cualquier construcción de lenguaje como for, while, do...while , en Javascript contamos con otras iteraciones especializadas en determinadas entidades :

```
Array.forEach(Function callback(Any element?, Number index?))
```

El método forEach se encuentra disponible en el prototipo de todos los arrays y algunos otros elementos que no son de la clase arrays pero aún así implementan esta interfaz. Toma un parámetro y es una función callback sincrónica que se ejecuta por cada elemento que exista dentro del array. El callback a su vez recibe por iteración al elemento por el cual está iterando y su índice dentro del array que lo contiene.

```
for( let i in obj){}
```

El bucle for...in es ideal para recorrer una matriz asociativa, osea un objeto literal cualquiera.

Funciones Flecha

La expresión de función flecha tiene una sintaxis más corta que una expresión de función convencional y no vincula sus propios this, arguments, super, o new.target. Las funciones flecha siempre son anónimas. Estas funciones son funciones no relacionadas con métodos y no pueden ser usadas como constructores.

```
var foo = function(){
      console.log("Foo!")
```

```
}

//Puede también expresarse como :
var foo = () => { console.log ("Foo!") }
```

Las funciones flecha tienen muchas variantes para poderlas escribir. Por el momento vamos a nombrar una muy útil:

```
var foo = function(unParametro){
      console.log("Foo!")
}

//Si la función flecha solo recibe un parámetro puede expresarse como:

var foo = unParametro => { console.log("Foo!") }
```

BOM y DOM

BOM Browser Object Model

El **BOM** ó **Browser Object Model** es la forma en la que Javascript representa al navegador en formato JSON. Es un objeto como cualquier otro con la particularidad de que es global, es decir que puede ser accedido en cualquier parte del programa usando su nombre window. Es tan global, que si quisiéramos usar alguna de las propiedades o métodos que tiene adentro, ni siquiera necesitamos mencionarlo para usarlas. Por ejemplo cada vez que usamos nuestra función log :

```
console.log("Soy Global!")
//Nunca decimos
window.console.log("Yo tambien soy global!")
```

De hecho, es tan global que todas las variables que escribimos sueltas en nuestros archivos van a ir a parar, a menos que les digamos lo contrario, a window! :

```
var a = true
console.log(a)               //true
console.log(window.a)   //true
```

Dentro de este objeto tenemos muchas propiedades que vamos a ir viendo a lo largo del curso pero para ir mencionando algunas que podríamos llegar a usar para escribir front-end podríamos mencionar cosas como :

```
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 dirección de la barra de navegación

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 Object Model

DOM o **Document Object Model** es la forma en la que Javascript representa HTML en formato JSON. Este no es un objeto global, pero si pertenece a uno (window). El Web API del DOM nos permite manipular nuestro HTML de la manera que queramos ya que dentro de su interfaz expuesta en la propiedad window.document tenemos una gran cantidad de métodos para llevar a cabo esta tarea :

Crear un nodo

```
document.createElement(tagName[, options]) => Element Object{}
```

La función document.createElement() crea un elemento HTML especificado por su tagName, o un HTMLUnknownElement si su tagName no se reconoce. Por lo tanto podríamos obtener una nueva etiqueta escribiendo algo como lo siguiente :

```
let h1 = document.createElement("h1")
```

Seleccionar un nodo existente

```
document.getElementById(id);
document.getElementsByClassName(names);
Element.getElementsByTagName(name);
Element.querySelector(selectors);
Element.querySelectorAll(selectors);
```

Los métodos de getElementById y getElementsByTagName están desde el comienzo del lenguaje prácticamente por lo que no necesitamos un fallback para usarlos en navegadores muy viejos como IE6+. Todos los demás se agregaron a partir de IE8+ 9.
Tener en cuenta que si optamos por usar los métodos nuevos, tenemos las ventaja de poder usar selectores de nodos mucho más avanzados que solo por ID o por nombre de etiqueta HTML.

Editar un nodo

Cada nodo, desde document en adelante, va a ser una referencia de alguna etiqueta de HTML y por consiguiente, los atributos de HTML se ven reflejados como las propiedades de cada nodo cada vez que seleccionamos o creamos uno nuevo; además existe otro lugar en donde se almacenan los atributos de HTML con su valor inicial y es la propiedad .attributes de cada nodo. La mayoría de los atributos de HTML se inicializan en cada nodo con el mismo nombre aunque algunos tienen un nombre distinto por cuestiones de nombres reservados. Entonces si quisiera modificar el atributo ID de HTML de algún nodo podría :

```
let h1 = document.createElement("h1")
h1.id = "titulo"
console.log(h1)   // <h1 id="titulo"></h1>
```

Hay propiedades que nos presentan alguna complejidad extra a veces . Podríamos mencionar la de el atributo CLASS y DATA-* . Si uno quisiera manipular las clases de una etiqueta de HTML tendría que usar la propiedad .className la cual guarda un string con el o las clases registradas que tenga esa etiqueta hasta ese momento. Por lo tanto, agregar o remover clases re volvería una tarea

demasiado tediosa para realizar a mano. Por lo que Javascript nos brinda facilidades como por ejemplo :

```
h1.classList.add("clase1")
h1.classList.remove("clase1")
h1.classList.toggle("clase1")
```

De esta forma tenemos una manera más conveniente de manipular clases. De esta misma forma también podemos observar el caso de los atributos data :

```
<h1 data-type = "title">Titulo</h1>
let h1 = document.createElement("h1")
console.log(h1.dataset)
h1.dataset.type = "nuevoTipo"
```

Para poder manipular este tipo de atributos también tenemos una interfaz JSON más funcional.

Agregar / Remover Nodos del DOM

Cada nodo por si solo tiene la habilidad de incorporar y remover nodos de su propio interior, es decir que todos van a tener un método que corresponde a cada operación :

```
element.appendChild(Child);
```

Este método nos permite agregar un nodo cualquiera dentro de otro que hayamos seleccionado previamente. Por ejemplo, si quisiéramos agregar una nueva etiqueta que creamos a una que ya teníamos escrita en HTML podríamos hacer lo siguiente :
En el HTML tendría una etiqueta con las siguientes características :

```
<div id= "cont"></div>

Y luego en mi script:

const cont = document.getElementById("cont")
const p = document.createElement("p")
p.innerText = "Lorem Ipsum"
cont.appendChild(p)
```

Si tuviera que agregar muchos elementos al mismo tiempo solo puedo hacerlos uno por uno, no admite ingreso de múltiples nodos en simultáneo.
Por otro lado si quisiera remover un nodo existente podría realizar algo como esto en el HTML :

```
<div id = "cont">
      <p>lorem ipsum</p>
</div>
```

Y algo como esto en el script :

```
const p = document.querySelector("p")
const parent = p.parentNode
parent.removeChild(p)
```

Eventos

Introducción

Los eventos son la ejecución de funciones como respuesta a una acción y se envían para notificar al código de cosas interesantes que han ocurrido. Cada evento está representado por un objeto que se basa en la interfaz Event, y puede tener campos y/o funciones personalizadas adicionales para obtener más información acerca de lo sucedido. Los eventos pueden representar cualquier cosa desde las interacciones básicas del usuario para notificaciones automatizadas de las cosas que suceden en el modelo de representación. Los eventos y el manejo de eventos proporcionan una técnica básica en JavaScript para reaccionar ante incidentes que ocurren cuando un navegador accede a una página web, incluidos eventos desde la preparación de una página web para visualizar, interactuando con el contenido de la página web, en relación con el dispositivo en el que el navegador se está ejecutando y de muchas otras causas, como la reproducción de secuencias de medios o el tiempo de animación.
Los eventos y el manejo de eventos se vuelven centrales para la programación web con la adición del idioma a los navegadores, acompañando un cambio en la arquitectura de renderizado de los navegadores desde el pedido, carga y procesamiento de páginas hasta el renderizado orientado a eventos basado en reflujo.
Inicialmente, los navegadores esperan hasta que reciban todos los recursos asociados con una página, para analizar, procesar, dibujar y presentar la página al usuario. La página mostrada permanece sin cambios hasta que el navegador solicita una nueva página. Con el cambio a la representación dinámica de la página, los navegadores realizan un bucle continuo entre el procesamiento, el dibujo, la presentación del contenido y la espera de un nuevo evento desencadenante. Los desencadenantes de eventos incluyen la finalización de la carga de un recurso en la red, por ejemplo, descargar una imagen que ahora se puede dibujar en la pantalla, la finalización de analizar un recurso por el navegador, por ejemplo, procesa el contenido HTML de una página, la interacción de un usuario con el contenido de la página, por ejemplo, hace clic en un botón.

Arquitectura de un evento

El sistema de eventos, en su núcleo, es simplemente un patrón de diseño de programación. El patrón comienza con un acuerdo sobre un tipo de evento y:

- El nombre String utilizado para el evento.
- El tipo de estructura de datos utilizada para representar las propiedades clave de ese evento.
- El objeto JavaScript que 'emitirá' ese evento.

El patrón es implementado por :

- Una función de JavaScript que toma como argumento la estructura de datos que se acordó.
- Registrando la función usando el nombre String con el objeto que emitirá el evento.

Se dice que la función es un "oyente"(listener) o un "manejador"(handler) con ambos nombres utilizados indistintamente. Este patrón se puede seguir fácilmente usando un código completamente personalizado, como se explica en el artículo sobre eventos personalizados. El patrón también es utilizado por los navegadores web modernos que definen muchos eventos emitidos en respuesta a la entrada del usuario o la actividad del navegador.
Los navegadores web modernos siguen el patrón del evento utilizando un enfoque estandarizado. Los navegadores usan como estructura de datos para las propiedades del evento, un objeto derivado del objeto EventPrototype. Los navegadores usan como método de registro para la función que manejará esas estructuras de datos un método llamado addEventListener que espera como

argumentos un nombre de tipo de evento de cadena y la función de controlador.
Finalmente, los navegadores definen una gran cantidad de objetos como emisores de eventos y definen una amplia variedad de tipos de eventos generados por los objetos.

Definiendo Eventos

En una primera instancia, podríamos definir un evento cualquiera asociando una etiqueta de HTML con el atributo correspondiente a un evento aceptado por esa etiqueta:

```
<button onclick =  "console.log('click!')">Clickeame</Button>
```

Ahora si le hacemos click al botón que aparece en la página y tenemos la consola de desarrollo abierta podremos ver un mensaje que dice “click!” tantas veces como le hayamos hecho click.
Como bien sabemos, la W3C nos dicen que cada lenguaje debería ser escrito en su propio entorno, con lo cual podemos extrapolar la misma función pero en un script externo:

```
var btn = document.querySelector("button")
btn.onclick = console.log("click!")
```

Ahora si recargamos la página nos damos cuenta que sin siquiera haber hecho click ya tenemos un mensaje en la consola y sobre todo que el evento ya no funciona.
Esto se debe a que Javascript determina que una función se va a ejecutar en la misma línea donde se mencionó si la misma lleva sus paréntesis de ejecución al lado. La función que tenemos escrita en el ejemplo anterior (console.log) tiene sus paréntesis de ejecución por lo que la misma se ejecutó instantáneamente. Si una función se ejecuta, en lugar de esta pasa a estar la expresión que la misma retorna, pero la función log no tiene retorno, por lo que vale undefined. Entonces es como si hubiéramos escrito:

```
console.log("click")
btn.onclick = undefined
```

De ahí podemos deducir el resultado obtenido.
Resulta que las funciones que le pasamos a un evento no tienen que estar ejecutadas sino que tenemos que pasarle la referencia a esa función, es decir la definición entera de una función.
Podemos lograr este objetivo solamente usando el nombre de la función o creando una función anónima en el mismo evento.

```
Function foo(){
      console.log("click")
}

btn.onclick = foo
btn.onclick = function(){
      console.log("click anónimo")
}
```

Si volvemos a correr este programa podemos notar ahora que vuelve a funcionar el vento pero no estamos viendo los dos click sino uno solo, el último asignado a la variable onclick del nodo.
Por este motivo, si bien podemos crear eventos de esta forma, podemos optar por registrar eventos a un nodo determinado sin utilizar sus propiedades internas ya que como vimos recién las mismas pueden ser sobreescritas.

Element.addEventListener

addEventListener() Registra un evento a un objeto en específico. El ObJeto específico puede ser un simple elemento en un archivo, el mismo documento , una ventana o un XMLHttpRequest.
Para registrar más de un eventListener, puedes llamar addEventListener() para el mismo elemento pero con diferentes tipos de eventos o parámetros de captura.

```
target.addEventListener(String tipo, Function listener[useCapture]);
```

Este método nos permite entonces registrar tantos eventos del mismo tipo como queramos al mismo elemento.

```
btn.addEventListener("click",foo)
btn.addEventListener("click",function(){
      console.log("click anonimo")
})
```

Si volvemos a correr el programa ahora podemos notar como ambos mensajes aparecen en consola.

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

Una referencia a un objeto que lanzó el evento. Es diferente de event.currentTarget donde el evento es llamado.

```
document.addEventListener("click", function(e){
      console.log(e.target)
})
```

Con las líneas de código anteriores podríamos enterarnos por consola de cada nodo que esté siendo clickeado en el DOM ya que los mismos no solo disparan su propio evento click, por más que no tenga un callback asignado, sino que además propagan el evento hasta el nodo superior, es decir <html> que en nuestro caso estáre presentado por la interfaz del DOM en document, por lo cual nos llegan todos los clicks de todos los nodos, a menos que estos hayan detenido su propagación con stopPropagation().
El target siempre es el mismo desde todos los Event, por lo que siempre tenemos referencia directa de que nodo inició la cadena de eventos:

```
<!-- index.html -->
<style>
```

```
#uno{
      width: 200px;
      height: 200px;
      background-color:red
}
#dos{
      width: 150;
      height: 150;
      background-color:blue
}
#tres{
      width: 100;
      height: 100;
      background-color:green
}

</style>
<div id="uno">
      <div id="dos">
            <div id="tres"></div>
      </div>
</div>

//index.js
function foo(e){ console.log(e.target,e.currentTarget)}
document.querySelector("#uno").addEventListener("click",foo)
document.querySelector("#dos").addEventListener("click",foo)
document.querySelector("#tres").addEventListener("click",foo)
```

Podemos observar en este otro ejemplo que obtenemos distintos resultados dependiendo que en posición del DOM estemos haciendo click.De cualquier manera, un uso frecuente de target es el de poder tener acceso directo al nodo que acaba de disparar el evento si tener que pedirlo por referencia de su variable ajena al ámbito local del callback.
Además de sus propiedades , los objetos Event tienen métodos que nos permiten entre otras cosas cancelar la propagación de una fase ó cancelar el comportamiento por defecto que tiene un evento.

stopPropagation

Es un método que nos sirve para detener la propagación de un evento desde el handler en donde se ejecuta :

```
document.addEventListener("click", function(){
      console.log("Click del DOM")
})
btn.addEventListener("click", function(e){
      e.stopPropagation()
      //...
})
```

Ahora podemos observar que el click que teníamos registrado para el DOM ya no se ve más en la consola, a menos que tengamos el evento en la fase capturing todavía, en cuyo caso o bien le cancelamos la propagación al DOM o volvemos los eventos a su fase bubbling.

preventDefault

Es un método que nos permite cancelar el comportamiento por defecto que pueda llegar a presentar un evento :

```
//index.html
<a href = "http://google.com">Ir a Google!</a>

//app.js
vara = document.querySelector("a")
      a.addEventListener("click", function(e){
      e.preventDefault()
      //...
})
```

De esta manera podemos interceptar el click de un elemento y realizar las operaciones que nosotros queramos en el DOM y no lo que el nodo haría por defecto, en este caso llevarnos a la ubicación de su atributo href.

Eventos Customizados

Javascript nos da la posibilidad de tener eventos customizados que no se ajusten con ningún evento predefinido y dispararlos en el momento en que queramos obteniendo todas las ventajas de las fases y cancelación de eventos en elementos del DOM. Para usar un evento customizado tenemos que :

1. Crear un evento
2. Asignárselo a un elemento
3. 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);
```

Eventos de elementos dinámicos

Muchas veces nos pasa que queremos registrar un evento en un elemento que aún no existe, es decir no se encuentra en el código fuente del DOM el cual evaluó el script en la carga de la página. Para solucionar este problema contamos con dos opciones:

● Tenemos acceso al lugar del código donde se crea ese elemento dinámico y podemos editar el programa para escribir el evento alrededor de esas líneas.

● Tenemos que registrar un evento a un elemento existente en el DOM al momento en que nuestro script se evalúa e interceptar la propagación que genera el target dinámico.

```
//index.html
<button>Crear!</button>
```

Teniendo en cuenta este HTML podemos entonces :

```
//app.js
var btn = document.querySelector("button")
btn.addEventListener("click", function(){
      var btn_dinamico = document.createElement("button")
      btn_dinamico.id = "dinamico"
      btn_dinamico.innerText = "dinamico!"
      btn_dinamico.addEventListener("click", function(){
            console.log("click dinámico")
      })
```

```
      document.body.appendChild(btn_dinamico)
})

Ó sino :

//app.js
var btn = document.querySelector("button")
btn.addEventListener("click", function(){
      var btn_dinamico = document.createElement("button")
      btn_dinamico.id = "dinamico"
      btn_dinamico.innerText = "dinamico!"
      document.body.appendChild(btn_dinamico)
})
document.addEventListener("click", function(e){
      if(e.target.id == "dinamico"){
            console.log("dinamico!")
      }
})
```

AJAX

Que es AJAX?

AJAX, acrónimo de ***A**synchronous **J**avaScript **A**nd **X**ML* (JavaScript asíncrono y XML), es una técnica de desarrollo web para crear aplicaciones interactivas o RIA (*Rich Internet Applications*). Estas aplicaciones se ejecutan en el cliente, es decir, en el navegador de los usuarios mientras se mantiene la comunicación asíncrona con el servidor en segundo plano. De esta forma es posible realizar cambios sobre las páginas sin necesidad de recargarlas, mejorando la interactividad, velocidad y usabilidad en las aplicaciones.

Ajax es una tecnología asíncrona, en el sentido de que los datos adicionales se solicitan al servidor y se cargan en segundo plano sin interferir con la visualización ni el comportamiento de la página, aunque existe la posibilidad de configurar las peticiones como síncronas de tal forma que la interactividad de la página se detiene hasta la espera de la respuesta por parte del servidor.

JavaScript es un lenguaje de programación (scripting language) en el que normalmente se efectúan las funciones de llamada de Ajax mientras que el acceso a los datos se realiza mediante *XMLHttpRequest*, objeto disponible en los navegadores actuales. En cualquier caso, no es necesario que el contenido asíncrono esté formateado en XML.

Ajax es una técnica válida para múltiples plataformas y utilizable en muchos sistemas operativos y navegadores dado que está basado en estándares abiertos como JavaScript y Document Object Model (DOM).

Tecnologías incluidas en Ajax

Ajax es una combinación de cuatro tecnologías ya existentes:

- XHTML (o HTML) y hojas de estilos en cascada (CSS) para el diseño que acompaña a la información.

- Document Object Model (DOM) accedido con un lenguaje de scripting por parte del usuario, especialmente implementaciones ECMAScript como JavaScript y JScript, para mostrar e interactuar dinámicamente con la información presentada.
- El objeto XMLHttpRequest para intercambiar datos de forma asíncrona con el servidor web. En algunos frameworks y en algunas situaciones concretas, se usa un objeto `iframe` en lugar del XMLHttpRequest para realizar dichos intercambios. PHP es un lenguaje de programación de uso general de script del lado del servidor originalmente diseñado para el desarrollo web de contenido dinámico también utilizado en el método Ajax.
- XML es el formato usado generalmente para la transferencia de datos solicitados al servidor, aunque cualquier formato puede funcionar, incluyendo HTML preformateado, texto plano, JSON y hasta EBML.

Ejemplo de Uso

```
<!DOCTYPE html>
<html>
    <head>
        <title>TODO supply a title</title>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
    </head>
    <body>
        buscar ID:<input type=text name=id id=id value=0 />
        <input type=button onclick="realizarProceso()" value=Calcula />
        <br>
        <span id=registros ></span>
        <script>
            var ajax;
            function realizarProceso() {
                url = 'http://localhost:8082/Parte05/ServletInscriptos?id=' +
                        document.getElementById('id').value;
                console.log('url=', url);
                ajax = new XMLHttpRequest();
                ajax.addEventListener('load', cargarValores);
                ajax.open('GET', url);
                ajax.send();
            }
            function cargarValores() {
                document.getElementById('registros').innerHTML = ajax.response;
            }
        </script>
    </body>
</html>
```