JAVASCRIPT FRONTEND AVANZADO
CLASE 2 : EVENTOS

EVENTOS

Qué es un evento?

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.[1]

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.

Douglas Crockford explica este cambio de manera efectiva en varias conferencias, especialmente su charla, Una API incómoda: la teoría del DOM, que muestra el cambio en el flujo desde el flujo original del navegador al navegador impulsado por el evento. El último enfoque cambia los últimos pasos de un flujo único a un ciclo perpetuo, donde la pintura y la espera y el manejo de la incidencia de nuevos eventos sigue. La innovación del enfoque dinámico permite que una página se represente parcialmente incluso cuando el navegador no ha terminado de recuperar todos los recursos; este enfoque también permite acciones impulsadas por eventos, que aprovecha JavaScript. (La charla está disponible en varias fuentes, incluido en https://www.youtube.com/watch?v=Y2Y0U-2qJMs). Actualmente, todos los entornos de ejecución para código JavaScript usan eventos y manejo de eventos. [2]

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, y
- 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, y
- 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 WHATWG y 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 [3] 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.

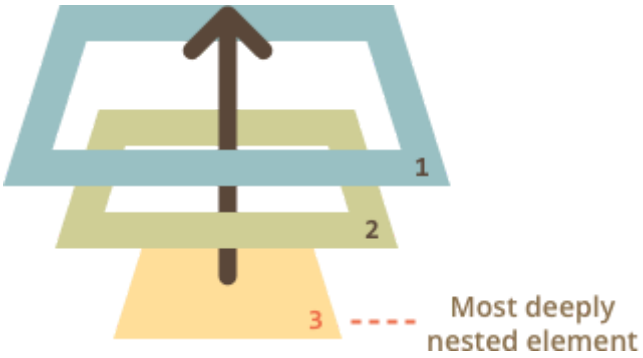
Usando esta dinámica de trabajo tenemos más facilidades de control de código como por ejemplo :

- Remover un handler identificado con nombre de un elemento
- Cambiar la fase en la que se ejecuta un evento

FASES

La fase de un evento se define como la forma en que el mismo se propaga a través del DOM. Por defecto todos los eventos vienen con una fase predeterminada pero podemos optar por cambiarlo de fase o bien cancelar cualquiera de las dos si no las necesitaremos. Las fases pueden ser :

- **Bubbling :** Ejecuta el handler del elemento target (el mismo que dispara el evento) y luego intenta disparar los handlers que haya registrados en sus nodos padres directos hasta llegar al elemento raíz osea el HTML(DOM)
- **Capturing :** Es la fase inversa a Bubbling. Dispara los handlers registrados del mismo tipo de evento arrancando por el elemento raíz y bajando hasta el target.



En la imagen anterior podemos observar la propagación por defecto bubbling que tiene un evento cualquiera

Por defecto los eventos ya se disparan en la fase **Bubbling** pero podemos cambiarlo de fase usando el último parámetro opcional del addEventListener.

```
document.addEventListener("click",function(){
     console.log("click DOM")
},true)
```

```
btn.addEventListener("click",function(){
     console.log("click boton")
},true)
```

De esta manera, si recargamos la página y le hacemos click al botón podremos notar que ahora primero se dispara el click registrado para el DOM, es decir la etiqueta HTML y luego el del target, es decir el elemento que disparó originalmente todo el evento de click: Nuestro botón.

A menos que no estemos usando las fases para construir alguna funcionalidad en particular, podremos siempre optar también por cancelar la fase. Para ello necesitamos hacer uso el objeto Evento construido por la misma arquitectura del lenguaje y enviado al handler del evento como primer parámetro, con lo cual es nuestra responsabilidad declarar una variable para poder hacer uso del mismo :

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

En el ejemplo anterior estamos declarando la variable "**e**" para capturar el objeto Evento representación del evento click que se disparó gracias a la acción del usuario al hacer click en el botón.

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. [4]

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.currentTargetdonde el evento es llamado durante la fase de bubbling or capturing :

```
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 está representado 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
```

```
var a = 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!")
     }
})
```

1. https://developer.mozilla.org/es/docs/Web/Reference/Events
2. https://developer.mozilla.org/en-US/docs/Web/Guide/Events/Overview_of_Events_and_Handlers
3. Definición de una función : Las funciones pueden retornar su definición solamente mencionando su puntero en memoria(nombre de variable) ó escribiendo una función anónima (Ej.: foo, function(){})
4. https://developer.mozilla.org/es/docs/Web/API/Event