

Desarrollo centrado en los/as usuarios/as

“La pregunta no es ¿qué queremos saber de la gente?, sino ¿qué quieren saber las personas de ellas mismas?”

Mark Zuckerberg— Programador (uno de los creadores y fundadores de Facebook)



Photo by [Cleo Vermij](#) on [Unsplash](#)

Cuando desarrollamos softwares, como developers nos formulamos dos preguntas básicas: ¿qué queremos que nuestro programa haga? ¿cómo queremos que lo haga? (otras tantas veces, esos requerimientos también vienen dados por un equipo de diseño o por nuestros clientes, ¡pero en cualquier caso estas dos preguntas aplican siempre!). Luego, escribimos el código. Pero, ¿de qué sirve todo eso si no ponemos a los/as usuario/as en el centro de nuestro proceso? ¿Acaso, cuando desarrollamos, no queremos satisfacer sus necesidades? Hay algo de nuestro alcance que está claro: los programas que generamos cobran sentido cuando estos entran contacto con usuarios/as reales.



Las interacciones en los sitios web son clave: desde el más mínimo click o la presión de una tecla. Por eso, es parte fundamental de esa tarea anticipar esos inputs para garantizar que haya una respuesta. En la bitácora anterior relacionamos los lenguajes HTML y CSS con Javascript y te mostramos qué puedes hacer sobre el DOM para modificar páginas web. En esta bitácora te invitamos a continuar aprendiendo sobre la relación entre HTML y Javascript.

¿Qué hacemos dentro del navegador?

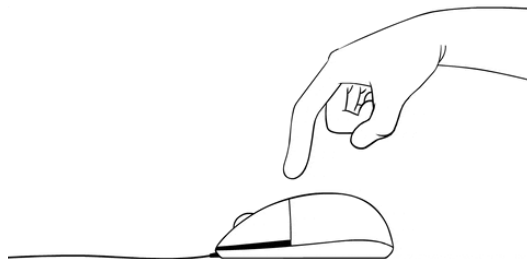
En este [artículo](#), Andréia Berto define lo siguiente:

*“La interacción de JavaScript con HTML se maneja a través de eventos que ocurren cuando el usuario o el navegador manipula una página, es decir, **los eventos forman las acciones u ocurrencias de una página web**, se usan en JavaScript para realizar acciones en la página en el navegador y cambiando el comportamiento predeterminado si es necesario.”*

En la bitácora 23 te anticipamos que nos encontramos con el paradigma de [Programación Orientada a Eventos \(POE\)](#) ¿lo recuerdas? ¿Sabes lo que significa? Si repasas la [Historia](#) de Javascript, verás que principalmente fue creado para **interceptar eventos que el/la usuario/a realiza sobre el sitio web**.

¿Pero qué es un evento? Un [evento](#) es una acción por parte de el/la usuario/a, al cual le definiremos un comportamiento.

Fíjate en los **formularios**: ¡en ellos podrás notar claramente los eventos! Actualmente el navegador, incluso, permite validar la información que completa el/la usuario/a antes de ser enviada al servidor y ante algún error le aparecerá una alerta de advertencia, es decir, un evento.



Los eventos que podemos capturar de los/as usuario/as pueden ser muchos: si hace un click con el mouse, si abandona una página web, si envía un formulario, si copia (*ctrl + c* o *cmd + c*) o corta (*ctrl + x* o *cmd + x*), si busca más información.

Los **handlers** son los encargados de "escuchar" los distintos inputs que el/la usuario/a realiza y de ejecutar las acciones que definimos para ese comportamiento en particular.

Para definir un handler hazte las siguientes preguntas en orden:

1. ¿Qué quiero escuchar?
2. ¿Dónde lo quiero escuchar?
3. ¿Que acción voy a hacer si lo escucho?

La función **addEventListener** nos permite poner **handlers** sobre un nodo y definir una función que se ejecutará cuando ese evento suceda.

Esas funciones son llamadas **callback** (más adelante profundizaremos sobre este tipo de funciones).

Su sintaxis es la siguiente:

```
nodo.addEventListener(evento, callback)
```

Ahora bien, respondamos estas preguntas y luego llevemoslo al código:

- ¿Qué quiero escuchar?
 - **Un click**
- ¿Donde quiero lo quiero escuchar?
 - **En un botón específico**
- ¿Que acción voy a hacer si lo escucho?
 - **Mostrar un mensaje**

Entonces, nuestro código queda de la siguiente manera:

```
let nodo = document.getElementById("btn");  
nodo.addEventListener("click", msj => alert("Hola"));
```

Aquí estamos viendo el evento *click* pero como te imaginarás existe una gran variedad de eventos, ¿quieres conocerlos todos? ¡Aquí tienes un [listado](#) de los eventos disponibles!



Entre los más conocidos podemos destacar:

- **click**: cuando el usuario realiza un click con el mouse
- **copy**: cuando el usuario copia (control + c o desde el mouse)
- **cut**: cuando el usuario corta (control + x o desde el mouse)
- **dblclick**: cuando el usuario realiza doble click con el mouse
- **focus**: cuando el usuario hace foco
- **keydown**: cuando presiona una tecla
- **keyup**: cuando suelta una tecla
- **mousedown**: cuando el usuario presiona el click del mouse
- **mouseup**: cuando el usuario suelta el click del mouse
- **mouseover**: cuando el usuario ingresa al área del elemento con el mouse
- **paste**: cuando el usuario pega un contenido (control + v o desde el mouse)

Cuando escribimos código para capturar las acciones del usuario/a lo que ocurre es que tanto la interfaz como lo que esperamos de ella va cambiando de acuerdo a estas interacciones, por lo que no es lo mismo un sitio web apenas el usuario ingresa que después de realizar N cantidad de interacciones. Lo que al inicio nos servía, tal vez después de 5 interacciones ya no sea útil. Por esta razón, muchas veces nos vemos forzados/as a eliminar los handlers y, para esto, Javascript nos provee del evento [removeEventListener](#).

Si sintaxis es similar al add

```
nodo.removeEventListener("click", callback);
```


Información sobre los eventos

A veces no solo necesitamos saber si un evento sucedió, sino además acceder a información más específica como por ejemplo: qué botón del mouse fue presionado, qué tecla, ¡incluso la ubicación desde donde fueron hechas estas acciones! **Al utilizar handlers, tendrás dentro de la función un objeto disponible que te proveerá más información sobre el evento que estás capturando.** Éste es el [objeto de evento](#).



¡Ten cuidado! La página [uniwebsidad](#) anticipa la primera dificultad: al momento de manipular los eventos de Javascript, los relacionados con el teclado son los más incompatibles entre diferentes navegadores y, por tanto, los más difíciles de manejar. Existen muchas diferencias entre los navegadores, los teclados y los sistemas operativos de los usuarios, principalmente debido a las diferencias entre idiomas.

¿Recuerdas el árbol de nodos? Para la mayoría de los eventos funciona igual, los controladores registrados en nodos con nodos hijos también recibirán la información de los eventos que sucedan sobre esos nodos. Muchas veces tanto el nodo padre como el hijo tienen `handlers` al mismo evento, por ejemplo para 'escuchar' un click. En estos casos, se ejecutan primero los handlers del hijo, y luego se ejecutan los handlers del padre.

 **TIP: se puede detener la propagación de eventos de los hijos hacia los padres.** Con el método `stopPropagation` debe ser llamado desde el objeto del evento. En la mayoría de los casos, en el objeto de evento también puedes encontrar la propiedad `target`, que refiere al nodo dónde se originó el evento.

Cierre

Llegas a este encuentro sabiendo la importancia de tener en el centro a usuarios/as en nuestro proceso de desarrollo. Ahora cuentas con herramientas y conocimientos sobre DOM con los que vas a poder generar los eventos de una página web para satisfacer sus necesidades lo mejor posible. ¿Te sientes preparado/a? Tómallo con calma, en este encuentro vamos a profundizar sobre el manejo de eventos.

¡Prepárate para el próximo encuentro!

Profundiza




[How Web Developers Respond to User Input](#)



[Everything about Event Handlers and Event Propagation](#)




Challenge

 Escribe un HTML que tenga la siguiente estructura:

Nombre: [input de texto] [botón]

Captura el click en el botón y da un alert que diga “Hola” + el nombre que escribieron en el input.

Challenge *opcional*

 Si el/la usuario/a presiona el botón y el input está vacío, realiza los siguientes pasos:

- Muestra el background del input en rojo
- Muestra un alert diciendo que el input está vacío
- Cuando el usuario comience a escribir vuelve el background del input a blanco