5. Interacción con el usuario. Eventos y formularios

DWEC
Mª Luz Sánchez Rubio

OBJETIVOS

- Reconocer las posibilidades de los lenguajes de marcas para capturar y gestionar los eventos producidos.
- Diferenciar los tipos de eventos que se pueden manejar.
- Crear código que capture utilice eventos.
- Reconocer las capacidades del lenguaje relativas a la gestión de formularios web.
- Validar formularios web utilizando expresiones regulares para facilitar los procedimientos.
- Utilización de cookies

1.- Acceso a elementos para establecer eventos

- Antes de poder asignar eventos a los elementos de una página web, es necesario **acceder a esos elementos desde JavaScript.**
- Actualmente existen varios métodos, aunque algunos son más modernos y flexibles que otros.

1.- Acceso a elementos para establecer eventos

Métodos tradicionales

Los siguientes métodos siguen funcionando y son últiles para entender cómo se comenzó a trabajar con el DOM

1. **getElementById(**id**)**:
    - Uso: Selecciona un elemento único basado en su atributo id
    - Ejemplo:

        *const boton = document.**getElementById**('miBoton');*

1.- Acceso a elementos para establecer eventos

Métodos tradicionales

2. **getElementsByTagName(**tagName**)**
   - **Uso:** Selecciona todos los elementos de un tipo específico de **etiqueta HtML** (como , <p>, <button>
   - **Ejemplo:**

     const botones=document.**getElementsByTagName**(‘button’)
3. **getElementsByName(**name**)**
   - **Uso:** Selecciona todos los elementos que comparten el mismo atributo name, normalmente en formularios
   - **Ejemplo:**

     const botonCampos=document.**getElementsByTagName**(‘usuario’)

1.- Acceso a elementos para establecer eventos

Métodos tradicionales

4. **getElementsByClassName(**className**)**
   - **Uso:** Selecciona todos los elementos con una clase específica
   - **Ejemplo:**

     const elementos=document.**getElementsByClassName**(‘miClase’)

Aunque estos métodos son válidos, **no permiten usar selectores CSS** ni devuelven siempre un objeto iterable moderno, por lo que su uso se ha reducido.

1.- Acceso a elementos para establecer eventos

Métodos modernos recomendados

1. **querySelector(selector)**
   - **Uso:** Selecciona **el primer elemento** del documento que coincida con el **selector CSS** especificado. **Id**, **clase**, **etiqueta**, o cualquier selector válido.
   - **Ejemplo:**

     const elemento=document.**querySelector**(‘#miClase’)

1.- Acceso a elementos para establecer eventos

Métodos modernos recomendados

1. **querySelectorAll(selector)**
   - **Uso:** Selecciona **todos los elementos** que coincidan con el **selector CSS** especificado y devuelve una **NodeList** (similar a un array) que se puede recorrer con métodos como **forEach().**

   - **Ejemplo:**

     const elementos=document.**querySelectorAll**(‘.miClase’)

2.- Posibilidades del Lenguaje de Marcas para capturar Eventos

- Los lenguajes de marcas como **HTML** permiten definir los elementos de una página web, mientras que **JavaScript** permite capturar los eventos producidos por la interacción del usuario con esos elementos.

- Estos eventos hacen que una página web sea dinámica e interactiva, mejorando la experiencia del usuario.

2.- Posibilidades del Lenguaje de Marcas para capturar Eventos

- **Eventos de Teclado**: Detectan cuando un usuario presiona una tecla o combina varias teclas.
  - ✓ **Ejemplo:** Capturar la entrada del usuario en un campo de texto (login, búsqueda).
  - ✓ **Evento común**: *keyup, keydown, input.*

- **Eventos de Ratón**: Capturan acciones como clics, doble clics o movimientos del cursor.
  - ✓ **Ejemplo**: Un botón que reacciona cuando el usuario hace clik.
  - ✓ **Evento común**: *click, dblclick, mouseover, mouseout, mousemove.*

2.- Posibilidades del Lenguaje de Marcas para capturar Eventos

- **Eventos de Carga de Página**: Detectan cuando de la página está disponible.
  - ✓ **Ejemplo:** Ejecutar código automáticamente una vez que la página se ha cargado por completo.
  - ✓ **Evento común**: *DOMContentLoaded, load.*

- **Eventos de Envío de formularios**: Capturan acciones como el envío o modificación de datos.
  - ✓ **Ejemplo**: Validar campos antes de permitir que se envíe un formulario.
  - ✓ **Evento común**: *submit, change, focus, blur, input, focusin, focusout.*

3.- Características del lenguaje para gestionar eventos

- Los lenguajes de programación como **JavaScript** proporcionan herramientas muy potentes para **gestionar eventos** y permitir la **interacción en tiempo real** con el usuario.

- La **gestión de eventos** en JavaScript se basa en la **programación orientada a eventos**, lo que significa que ciertas funciones (llamadas manejadores de eventos) se ejecutan automáticamente **en respuesta a una acción del usuario** o del sistema (por ejemplo, un clic, una pulsación de tecla o la carga de la página).

3.- Características del lenguaje para gestionar eventos

1. **addEventListenner()** permite asociar una función (callback) a un evento específico sobre un elemento del DOM. Cada vez que se produzca el evento, la función se ejecutará.

   Ejemplo:

```
boton.addEventListener('click', mostrarAlerta);
boton.addEventListener('click', ()=>{
  alert('Has hecho clic en el botón');
});
```

   Ventajas:

   - Permite asignar múltiples funciones al mismo evento.
   - Separa la lógica JavaScript del código HTML, mejorando la organización.
   - Permite especificar opciones avanzadas como **once** (ejecutar solo una vez) o capture (fase de captura)

```
botón.addEventListener('click', mostrarAlerta, {once : true});
```

3.- Características del lenguaje para gestionar eventos

2. **removeEventListener():** Elimina una función previamente asociada a un evento con **addEventListener().**
   Es útil cuando se desea desactivar un comportamiento temporal o evitar duplicar acciones.

   Ejemplo:

   boton.**removeEventListener**('click', mostrarAlerta);

4.- Diferenciación de tipos de eventos que se pueden manejar

- En el desarrollo web, los **eventos** son acciones o sucesos que ocurren en la página, normalmente como resultado de **interacción del usuario** o de un proceso del sistema.

- **JavaScript** permite **captura y reaccionar** ante esos eventos mediante funciones llamadas manejadores de eventos, que permiten crear interfaces dinámicas e interactivas.

4.- Diferenciación de tipos de eventos que se pueden manejar

Principales tipos de eventos:

1. **Eventos de ratón:**
   - Se disparan cuando el usuario interactúa con el ratón o dispositivo apuntador.
   - **Ejemplos:**
     - **click:** cuando el usuario hace clic sobre un elemento.
     - **dblclick:** cuando realiza un doble clic**.**
     - **mouseover :** El puntero entra en un elemento.
     - **mouseout**: El puntero sale del elemento.
     - **mousemove**: El puntero se mueve dentro del elemento.
     - **contextmenu**: al hacer clic derecho (abri menú contextual

```
document.querySelector(‘#miBoton').addEventListener('click', function() {
        alert(‘Has hecho doble clic en el botón!’);
}
```

4.- Diferenciación de tipos de eventos que se pueden manejar

Principales tipos de eventos:

2. **Eventos de teclado:**
   - Se activan cuando el usuario presiona o suelta una tecla. Pueden utilizarse para validar formularios, atajos de teclado o accesibilidad.
   - **Eventos:**
     - **keydown:** cuando una tecla es presionada
     - **keyup:** cuando una tecla se suelta**.**
     - **input:** Se dispara cada vez que cambia el valor de un campo.

```
document.addEventListener('keydown', (e) =>{
   console.log(`Tecla presionada:  ${e.key}`);
});
```

4.- Diferenciación de tipos de eventos que se pueden manejar

Principales tipos de eventos:

3. **Eventos de carga:**
   - Se disparan cuando una página o un recurso ha sido completamente cargado. Son útiles para ejecutar código **una vez que el DOM está disponible**
   - **Eventos:**
     - **DOMContentLoaded:** cuando el DOM está listo (sin esperar imágenes ni estilos).

```
document.addEventListener('DOMContentLoaded',  () =>{
   console.log('La página está lista');
});
```

4.- Diferenciación de tipos de eventos que se pueden manejar

Principales tipos de eventos:

4. **Eventos de formulario:**
   - Se disparan durante la interacción del usuario con elementos de formulario, como entradas de texto, selectores o botones de envío.
   - **Eventos:**
     - **submit:** cuando un formulario se envía.
     - **change:** cuando el valor de un campo cambia.
     - **focus**: cuando un campo obtiene el foco.
     - **blur**: cuando un campo pierde el foco.
     - **input:** se ejecuta en tiempo real cada vez que cambia el valor de un campo.

```
document.querySelector(‘#miFormulario').addEventListener('submit', (e)=> {
        e.preventDefault();    //evita el envío
        alert('Formulario enviado correctamente');
});
```

4.- Diferenciación de tipos de eventos que se pueden manejar

Principales tipos de eventos:

5. **Eventos de portapapeles:**
   - Permiten detectar o controlar las acciones de copiar, cortar o pegar texto.
   - **Eventos:**
     - **copy**
     - **cut**
     - **Paste**

```
document.querySelector(‘#nombre').addEventListener(‘paste', ()=> {
        console.log(‘Texto pegado en el campo');
});
```

4.- Diferenciación de tipos de eventos que se pueden manejar

Principales tipos de eventos:

5. **Eventos de ventana y scroll (Window Events):**
   - Relacionados con el tamaño de la ventana, el desplazamiento ola visibilidad de la página.
   - **Eventos:**
     - **resize:** cuando se cambia el tamaño de la ventana.
     - **scroll:** cuando el usuario hace scroll.
     - **visibilitychange:** cuando la pestaña cambia de estado (activa/inactiva).

```
window.addEventListener('scroll', ()=> {
        console.log('El usuario está haciendo scroll');
});
```

4.- Diferenciación de tipos de eventos que se pueden manejar

Principales tipos de eventos:

6. **Eventos de orientación del dispositivo:**
   - Relacionados con la orientación física del dispositivo (Smartphone o Tablet) usando giroscopio y acelerómetro.
   - **Eventos:**
     - **deviceorientation:** cuando cambia la orientación del dispositivo.
     - **devicemotion:** cuando cambia la aceleración o rotación.

```
window.addEventListener('devicemotion', (e)=> {
        console.log(`Aceleración x: ${e.acceleration.x}, y:
        ${e.acceleration.y}, z: ${e.acceleration.z}`);});
```

4.- Diferenciación de tipos de eventos que se pueden manejar

Principales tipos de eventos:

7. **Eventos de puntero:**
   - Relacionados con la interacción del usuario mediante mouse, táctil o stylus.
   - **Eventos:**
     - **pointerdown:** cuando se presiona un puntero sobre un elemento.
     - **pointermove:** cuando se mueve un puntero sobre un elemento.
     - **pointerup:** cuando se suelta un puntero sobre un elemento.
     - **pointerenter /pointerleave:** cuando un puntero entra o sale de un elemento.

```
box.addEventListener('pointerdown', (e) => {
  console.log('Pointer down:', e.pointerType, e.clientX, e.clientY);
});
```

5.- Capacidades del lenguaje para gestionar formularios

- **HTML** y ***JavaScript*** son las herramientas clave para crear y gestionar formularios web permitiendo la interacción con el usuario y el procesamiento de datos.

- **Características clave:**

1. **Captura y envío de datos:**
   - ✓ **HTML** define la estructura del formulario como <form>, <input>, <select>, etc., que permiten capturar datos del usuario.
   - ✓ **JavaScript** se encarga de controlar y enviar los datos dinámicamente.

```
const form = document.querySelector('#form');
form.addEventListener('submit', e => {
  e.preventDefault();
  const datos = new FormData(form);
  //envío datos
});
```

5.- Capacidades del lenguaje para gestionar formularios

2. **Validación de datos.**
   - ✓ **Validación nativa (HTML5).** Usa atributos como **required, pattern, minlength, type=email**, etc.

```
<input type="email" required>
```

   - ✓ **Validación con la Constraint Validation API:** Esta API permite comprobar si un campo es válido y mostrar mensajes personalizados.

```
const email = document.querySelector('#email');

if (!email.checkValidity()) { //verifica si el campo cumple todas las reglas de validación HTML5
  email.reportValidity(); // Muestra el mensaje de error nativo del navegador
}
```

5.- Capacidades del lenguaje para gestionar formularios

3. **Manejo de errores:**
   - ✓ **JS** permite informar al usuario sobre errores de validación de forma visual y personalizada, mejorando la experiencia de uso. Se recomienda destacar los errores **visualmente** (por ejemplo, con borde rojo o mensaje junto al campo):

```
const campo = document.querySelector('#nombre');

campo.addEventListener('input', () => {
  if (campo.value.trim() === '') {
    campo.classList.add('error');
    campo.setCustomValidity('El campo no puede estar vacío');
  } else {
    campo.classList.remove('error');
    campo.setCustomValidity('');
  }
});
```

6.- Validación de formularios utilizando eventos.

La validación de formularios es crucial para asegurar que los datos enviados por los usuarios sean correctos y estén en el formato adecuado. Usando eventos de JavaScript, se puede verificar y controlar la entrada del usuario antes que el formulario sea enviado.

Eventos clave utilizados para validar formularios:

1. **submit: (**envío del formulario**):** es ideal para realizar una **validación general** antes que los datos se envíen.

```
const form = document.querySelector('#registro');

form.addEventListener('submit', e => {
  if (!form.checkValidity()) {       // Comprueba si algún campo es inválido
    e.preventDefault();              // Evita el envío del formulario
    form.reportValidity();           // Muestra mensajes nativos del navegador
  } else {
    alert('Formulario válido y enviado correctamente');
  }
});
```

6.- Validación de formularios utilizando eventos.

Eventos clave utilizados para validar formularios:

2. **blur** (cuando el campo pierde el foco): permite validar cada campo cuando el usuario termina de escribir en él .

```
email.addEventListener('blur', () => {
 if (!email.checkValidity()) {
   email.reportValidity();
 }
});
```

3. **Input:** se dispara cada vez que el usuario cambia el contenido del campo, ya sea escribiendo, borrando o pegando texto.

```
telefono.addEventListener('input', () => {
 const valido = /^[0-9]{9}$/.test(telefono.value);
 telefono.setCustomValidity(valido ? '' : 'El número debe tener 9 dígitos');
 telefono.reportValidity();
});
```

6.- Validación de formularios utilizando eventos.

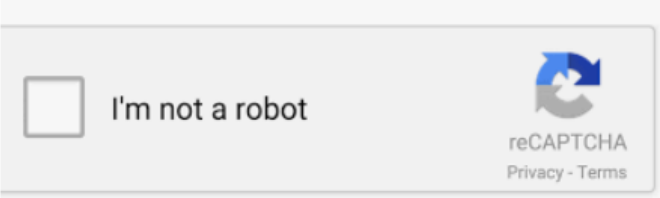
Buenas prácticas en validación:

- ✓ Usar los **atributos HTML5** (*required, type, pattern, min, max, etc.*)
- ✓ Completar con **JavaScript** para mensajes personalizados.
- ✓ Evitar abusar de **alert()**; usar mensajes cerca de los campos.
- ✓ Validar **también en el servidor** (JavaScript solo protege el cliente).
- ✓ Proporcionar mensajes claros y accesibles.

7.- Implementación de reCAPTCHA en formularios web.

La seguridad en los formularios web es fundamental para evitar el envío automático de datos por bots o scripts maliciosos.

Google reCAPTCHA es un servicio de seguridad gratuito proporcionado por Google que ayuda **a verificar que el usuario es humano**, bloqueando el spam y los intentos de acceso automatizados.



7.- Implementación de reCAPTCHA en formularios web.

Características Principales de reCAPTCHA:

1. **Protección contra bots:**
   - ✓ Distingue entre interacciones humanas y automáticas mediante desafíos visuales o de comportamiento.

2. **Versión 2 ("No soy un robot"):**
   - ✓ Los usuarios deben hacer clic en una casilla de verificación ("No soy un robot") y, a veces, resolver un desafío visual (seleccionar imágenes específicas).

3. **Versión 3 (Invisible reCAPTCHA):**
   - ✓ No requiere interacción del usuario. Funciona en segundo plano y asigna una puntuación basada en el comportamiento del usuario (0-10).

4. **Versión Enterprise:**
   - ✓ Ofrece protección avanzada para sitios web con tráfico empresarial o de gran volumen, con integraciones adicionales.

7.- Implementación de reCAPTCHA en formularios web.

Cómo funciona reCAPTCHA

- ✓ Cuando un usuario interactúa con un formulario protegido por reCAPTCHA, el sistema evalúa si es un usuario legítimo.
- ✓ Dependiendo de la versión (v2 o v3), se puede requerir una acción activa (clic o resolución de un desafío) o se ejecuta en segundo plano sin que el usuario lo note.
- ✓ Si el usuario pasa la verificación, el formulario se envía. De lo contrario, se bloquea el envío para evitar abuso.

8.- Uso de expresiones regulares para validar formularios.

Las expresiones regulares (RegExp) permiten validar de manera eficiente el formato de los datos en formularios. Estas pueden ser escritas de dos maneras en JavaScript: utilizando **límites** (//) o el **constructo**r (new RegExp). Ambas formas permiten verificar si los datos introducidos coinciden con un patrón específico.

Características clave del uso de expresiones regulares:

1. **Verificación precisa de formatos:**
   - ✓ Las RegExp verifican si un campo de texto tiene el formato correcto (ej. correos, contraseñas)
2. **Reducción de errores humanos:**
   - ✓ Detectan errores en la entrada antes de procesar los datos, garantizando que los datos sean válidos.
3. **Validación compleja en una sola línea:**
   - ✓ Las RegEx permiten validar formatos complejos, como fechas, contraseñas seguras y números de teléfono.

8.- Uso de expresiones regulares para validar formularios.

Métodos principales para trabajar con RegExp:

Método	Descripción	Ejemplo
`.test(cadena)`	Comprueba si la cadena cumple el patrón (devuelve `true` o `false`).	`/^[0-9]{9}$/.test('123456789')` → `true`
`.match(patrón)`	Devuelve las coincidencias encontradas.	`'abc123'.match(/[0-9]/g)`
`.replace(patrón, nuevo)`	Sustituye coincidencias por otro texto.	`'abc123'.replace(/[0-9]/g, '*')` → `'abc***'`

8.- Uso de expresiones regulares para validar formularios.

Ejemplo. Validar número de teléfono (9 dígitos):

```
// Usando //
const telefonoRegex = /^[0-9]{9}$/;

// Usando new RegExp()
const telefonoRegex = new RegExp('^[0-9]{9}$');

// Validación
console.log(telefonoRegex.test("123456789"));  // true
```

9.- Gestión de cookies, localStorage y sessionStorage en JavaScript.

En el desarrollo web moderno, JavaScript permite **almacenar datos del usuario directamente en el navegador del usuario**, lo que facilita recordar preferencias, mantener sesiones o guardar información temporal sin depender constantemente del servidor.

Existen tres métodos principales en JavaScript: **cookie**s, **localStorage** y **sessionStorage**. Cada uno tiene su propia finalidad, capacidad y nivel de persistencia.

Métodos para almacenar datos en el navegador:

1. **Cookies (document.cookie):**
   - ✓ Almacenan pequeños fragmentos de información (máximo 4 KB) que pueden ser enviadas automáticamente al servidor con cada solicitud HTTP. Son ideales para mantener sesiones o recordar configuraciones entre visitas.
   - ✓ Tienen una fecha de expiración y son accesibles desde cualquier página del mismo dominio.
   - ✓ **Ventajas**: compatibles con solicitudes HTTP, permitiendo enviar cookies al servidor.
   - ✓ **Limitaciones:** tamaño limitado y menos seguro que otras opciones de almacenamiento local.

9.- Gestión de cookies, localStorage y sessionStorage en JavaScipt.

Métodos para almacenar datos en el navegador:

2. **locaStorage**
   - ✓ Permiten almacenar datos sin límite de tiempo, es decir, la información se mantiene incluso después de cerrar el navegador.
   - ✓ Tiene una capacidad mayor de (5MB) que las cookies.
   - ✓ Los datos son accesibles en todas las pestañas y ventanas del mismo dominio.
   - ✓ **Ventajas**: Almacenamiento persistente y seguro para grandes cantidades de datos.
   - ✓ **Limitaciones:** Solo accesible desde el navegador (no se envía al servidor)

9.- Gestión de cookies, localStorage y sessionStorage en JavaScipt.

Métodos para almacenar datos en el navegador:

3. **sessionStorage**
   - ✓ Similar a **localStorage**, pero los datos se eliminan cuando la sesión del navegador termina (pestaña o ventana se cierra).
   - ✓ Útil para almacenar datos que solo son necesarios durante una sesión.
   - ✓ **Ventajas**: Ideal para datos temporales que solo deben estar disponibles durante una sesión
   - ✓ **Limitaciones:** Los datos no persisten después de cerrar la pestaña o ventana.

9.- Gestión de cookies, localStorage y sessionStorage en JavaScipt.

Comparación entre Cookies, localStorage y sessionStorage

Característica	Cookies	localStorage	sessionStorage
Tamaño máximo	~4KB	~5MB	~5MB
Duración	Definida por la expiración	Persistente	Solo durante la sesión
Acceso	Cliente y Servidor	Solo Cliente	Solo Cliente
Envío al Servidor	Sí (en cada solicitud)	No	No
Uso típico	Autenticación, Preferencias de usuario	Datos persistentes (configuración, estado)	Datos temporales (carritos, sesión actual)