

Desarrollo Web en Entorno Cliente

Unidad 5:
Interacción con el usuario. Eventos y
Formularios.

Objetivos.

- Capturar y gestionar eventos.
- Conocer los diferentes tipos de eventos.
- Generar código que capture y gestione eventos.
- Conocer las capacidades del lenguaje para gestionar formularios web.
- Validar formularios web utilizando eventos y expresiones regulares.
- Probar y documentar el código.

Contenido

1	Modelo de gestión de eventos.	4
1.1	Tipos de Eventos	5
1.2	Manejadores de eventos	6
1.3	El método addEventListener().	8
1.4	El objeto event.	15
1.5	Eventos de ratón.	19
1.6	Eventos de teclado.	21
1.7	Eventos HTML	22
1.8	Eventos del DOM.....	23
2	Utilización de formularios.	24
2.1	Utilización de formularios desde JavaScript.	27
3	Validación y envío de formularios.....	28
3.1	Algunos ejemplos de funciones de validación de datos	33
4	Expresiones regulares.	35
4.1	Métodos en JavaScript para trabajar con expresiones regulares.....	37
4.2	Validar un formulario con expresiones regulares.....	38

1 Modelo de gestión de eventos.

En la **programación orientada a eventos**, las aplicaciones esperan sin realizar ninguna tarea a que se produzcan un **evento**. Una vez producido, ejecuta alguna tarea asociada a ese evento y al terminar, el script o programa vuelve al estado de espera.

Los eventos de JavaScript permiten la interacción entre aplicaciones JavaScript y los usuarios. Cada vez que se pulsa un botón, o una tecla se produce un evento, también se puede producir un evento sin que intervenga el usuario, por ejemplo, cada vez que se carga una página.

En el modelo orientado a eventos, el ciclo de vida de una aplicación puede describirse en las siguientes fases:

1. Estado de espera (Idle): La aplicación está inactiva, sin realizar ninguna acción, esperando que ocurra un evento.
2. Producción del evento: Se genera un evento que puede ser causado por:
 - Interacciones del usuario, como hacer clic en un botón, mover el ratón o presionar una tecla.
 - Eventos del sistema, como la carga de una página, el redimensionamiento de la ventana o la finalización de una solicitud HTTP.
3. Ejecución de una tarea asociada al evento: Una vez que el evento se ha producido, la aplicación ejecuta el código asociado a ese evento, que es el manejador de eventos o event handler. Este código define lo que debe hacer la aplicación en respuesta a ese evento.
4. Vuelta al estado de espera: Una vez que el código asociado al evento se ha ejecutado, la aplicación vuelve a su estado de espera hasta que ocurra un nuevo evento.

Este proceso es asíncrono y, en lugar de que la aplicación realice tareas constantemente, el sistema de eventos maneja las acciones bajo demanda, optimizando el uso de recursos.

1.1 Tipos de Eventos

Los eventos en JavaScript no son solo resultado de la interacción del usuario. Existen varios tipos de eventos que pueden desencadenar acciones:

1. **Eventos de usuario:** Son los eventos más comunes que ocurren cuando un usuario interactúa con la interfaz:
 - **Eventos de ratón:** *click, dblclick, mousemove, mousedown, mouseup*, etc.
 - **Eventos de teclado:** *keydown, keyup, keypress*.
 - **Eventos de interfaz de usuario:** *scroll, resize*.
2. **Eventos del sistema o del navegador:** Estos eventos no requieren intervención directa del usuario.
 - **Eventos de la ventana:** *Load* (cuando la página ha sido completamente cargada), *beforeunload* (antes de que se cierre la página), *error* (cuando ocurre un error en la página o en la carga de un recurso).
 - **Eventos de tiempo:** *setTimeout, setInterval* permiten programar acciones que ocurren después de un periodo de tiempo específico.
3. **Eventos personalizados:** Los desarrolladores pueden crear sus propios eventos personalizados utilizando *CustomEvent*. Estos pueden dispararse de acuerdo con las necesidades específicas de la aplicación.

Más información: http://www.w3schools.com/tags/ref_eventattributes.asp

1.2 Manejadores de eventos

Una vez que se produce un **evento**, la aplicación responde ejecutando cierto **código asociado o función**. Este código o función se denomina **manejador de eventos** (*event handler*), las funciones externas que se definen para responder a los eventos se denominan **funciones manejadoras**.

Podemos asignar un manejador de evento para un elemento HTML utilizando diferentes técnicas:

1. **Mediante el atributo HTML:** Es la forma más sencilla de incluir un manejadores de eventos, pero no es modular ni eficiente para aplicaciones.

```
<button onclick="alert('Has pulsado el botón validar')">Validar</button>
```

2. **Asignación mediante propiedades JavaScript:** Cada elemento DOM tiene propiedades para eventos como onclick, onchange, onsubmit, etc. Se puede asignar una función a estas propiedades.

```
let boton = document.querySelector('button');
boton.onclick = function() {
    alert('Has pulsado el botón');
};
```

3. **addEventListener():** Este es el método recomendado y más versátil para gestionar eventos, ya que permite asignar múltiples eventos a un solo elemento sin sobrescribir los anteriores.

```
let boton = document.querySelector('button');
boton.addEventListener('click', function() {
    alert('Has pulsado el botón');
});
```

Al crear páginas web debemos separar los **contenidos (HTML)** de la **presentación (CSS)** y de la **programación (JavaScript)**. Mezclar JavaScript y HTML complica el código fuente, dificulta su mantenimiento. Para definir manejadores de eventos en JavaScript de una forma que mantenga el código limpio, modular y separado del HTML, se recomienda separar el contenido, la presentación y el comportamiento de una página web en tres capas bien definidas:

Capa	Descripción	Tecnología
Contenido	Define la estructura y el contenido de la página web (textos, imágenes, enlaces, etc.)	HTML
Presentación	Controla la apariencia del contenido, incluyendo estilos, colores, tipografía, y posicionamiento.	CSS
Comportamiento	Añade interactividad y dinámica al contenido mediante la manipulación del DOM y la respuesta a eventos del usuario.	JavaScript

Esta organización en capas mejora la estructura, facilita la legibilidad y el mantenimiento, permitiendo que cada archivo cumpla un propósito único. Al actualizar, por ejemplo, el diseño, solo se debe modificar el CSS; si es necesario ajustar la interactividad, únicamente el archivo JavaScript.

Separación de JavaScript y HTML para manejar eventos.

Al manejar eventos, debemos evitar prácticas como el uso de atributos de evento directamente en el **HTML** (**onclick**, **onmouseover**, etc.). Aunque esta técnica funciona, mezcla el contenido y el comportamiento, haciendo que el código sea difícil de mantener y menos modular.

```
<!-- No recomendado: JavaScript dentro del HTML -->
<button onclick="mostrarMensaje()">Haz clic aquí</button>
<script>
    function mostrarMensaje() {
        alert('¡Hola!');
    }
</script>
```

Donde: **onclick="mostrarMensaje()"** está directamente en el HTML, lo que mezcla la estructura y el comportamiento. Haciendo que encontrar y cambiar el código de JavaScript dentro de un HTML extenso puede ser complejo y poco eficiente.

Definir manejadores de eventos con **addEventListener()**.

Este método permite asignar eventos a los elementos del DOM desde el archivo JavaScript, sin modificar el HTML. De este modo, todo el comportamiento se centraliza en el archivo de JavaScript, facilitando el mantenimiento y la reutilización.

```
<!-- html -->
<button id="btnAceptar">Aceptar</button>

// javascript
document.addEventListener('DOMContentLoaded', function () {
    const boton = document.getElementById('btnAceptar');

    boton.addEventListener('click', function () {
        alert('Has pulsado el botón aceptar');
    });
});
```

Donde: **addEventListener()** asigna el evento **click** al botón con el **id="miBoton"**. Toda la lógica está en el archivo JavaScript, dejando el HTML limpio y semántico.

1.3 El método addEventListener().

El método **addEventListener()** es una forma moderna y flexible de manejar eventos en JavaScript. Permite registrar un manejador de eventos a un objeto del Document Object Model (DOM), como un elemento HTML, un documento o incluso la ventana del navegador, sin sobrescribir otros manejadores de eventos previamente asignados. Este método es ampliamente utilizado debido a su flexibilidad, ya que permite añadir múltiples manejadores de eventos al mismo elemento y controlar diversas fases de propagación del evento.

Sintaxis:

```
elemento.addEventListener(tipoDeEvento, funcionManejadora [, opciones]);
```

- **elemento:** El elemento DOM al que se asocia el evento.
- **tipoDeEvento:** Una cadena de texto que representa el nombre del evento (*click*, *keydown*, *submit*, etc.).
- **funcionManejadora:** La función que se ejecutará cuando ocurra el evento. Puede ser una función anónima, una función flecha o una función previamente definida.
- **opciones (opcional):** Un tercer argumento que puede ser un valor booleano o bien un objeto de opciones. Permite controlar ciertos aspectos del comportamiento del evento, como si se captura o no el evento durante la fase de captura o de burbujeo.

Ejemplos:

```
let boton = document.getElementById('btnCancelar');
boton.addEventListener('click', function() {
    console.log('Has pulsado cancelar');
});

function mostrarMensaje() {
    alert('¡Hola, has pulsado el botón!');
}
let boton = document.querySelector('button');
boton.addEventListener('click', mostrarMensaje);

// ejemplo varios manejadores
let boton = document.querySelector('button');

boton.addEventListener('click', function() {
    console.log('Manejador de evento 1');
});

boton.addEventListener('click', function() {
    console.log('Manejador de evento 2');
});
```

Opciones del método addEventListener()

El tercer parámetro de `addEventListener()` puede ser un valor booleano o un objeto de opciones. Este argumento permite ajustar el comportamiento del evento en cuanto a la fase de propagación, si debe ejecutarse una única vez, entre otros aspectos.

- **Booleano:** el tercer argumento de `addEventListener()` puede ser un valor booleano. Si es `true`, se indica que el manejador debe ejecutarse en la fase de **captura**. Si es `false`, se ejecuta en la fase de **burbujeo**.

```
// Se ejecuta en la fase de captura
elemento.addEventListener('click', function() {
    console.log('Captura');
}, true);

// Se ejecuta en la fase de burbujeo (por defecto)
elemento.addEventListener('click', function() {
    console.log('Burbujeo');
}, false);
```

- **Objeto de Opciones:** Con el tiempo, se introdujo un objeto de opciones para tener más control sobre los eventos. Este objeto puede contener las siguientes propiedades:

- **`capture`:** Si el evento debe manejarse en fase de **captura** (`true`) o de **burbujeo** (`false`, por defecto).
- **`once`:** Si es `true`, el manejador de eventos se ejecutará solo **una vez** y se eliminará automáticamente. Si no necesitamos que un evento se dispare varias veces.
- **`passive`:** Si es `true`, indica que el manejador de eventos **no llamará** a `preventDefault()`. Mejorando el rendimiento en algunos eventos como `scroll` o `touchmove` en dispositivos móviles.
- **`signal`:** Permite asociar el evento a un **`AbortController`** para eliminar el evento de forma programática.

Ejemplos:

```
let boton = document.getElementById('miBoton');

// usando el objeto de opciones
boton.addEventListener('click', function() {
    console.log('Este manejador se ejecutará solo una vez');
}, { once: true });

boton.addEventListener('click', function() {
    console.log('Este manejador se ejecuta en burbujeo, como
siempre');
}, { capture: false });
```

```
// ejemplo con passive
window.addEventListener('scroll', function() {
    console.log('Desplazamiento detectado');
}, { passive: true });
```

Eliminación de un manejador de eventos: removeEventListener()

Para eliminar un manejador de eventos previamente añadido con `addEventListener()`, se utiliza el método `removeEventListener()`. Es importante que la referencia a la función sea la misma que la usada en `addEventListener()`, ya que las funciones anónimas no pueden ser eliminadas directamente.

```
function mostrarMensaje() {
    alert('Botón pulsado');
}

let boton = document.querySelector('button');

// Añadir el evento
boton.addEventListener('click', mostrarMensaje);

// Quitar el evento
boton.removeEventListener('click', mostrarMensaje);
```

Si usamos una **función anónima** en `addEventListener()`, no es posible eliminar ese manejador de eventos, ya que no se tiene una referencia directa a esa función:

```
// Esto no funcionará
boton.addEventListener('click', function() {
    alert('Botón clickeado');
});

boton.removeEventListener('click', function() {
    alert('Botón clickeado');
});
```

Ejemplos de asignación de eventos usando addEventListener().

Asignar eventos a elementos con un selector.

```
<!-- html -->
<button class="boton">Botón 1</button>
<button class="boton">Botón 2</button>
<button class="boton">Botón 3</button>

// javascript
document.addEventListener('DOMContentLoaded', function () {
    const botones = document.querySelectorAll('.boton');
    botones.forEach(boton => {
        boton.addEventListener('click', function () {
            alert('Botón clickeado');
        });
    });
});
```

Donde: se asigna el evento *click* a varios botones con la clase *.boton*, sin modificar el HTML de cada botón.

Asignar eventos dinámicamente.

A veces, necesitamos asignar eventos a elementos creados dinámicamente en el DOM. Esto se puede hacer asignando el evento al contenedor del elemento y usando la delegación de eventos.

```
<!-- html -->
<div id="contenedor">
    <button class="boton">Botón 1</button>
</div>

// javascript
document.addEventListener('DOMContentLoaded', function() {
    const contenedor = document.getElementById('contenedor');
    contenedor.addEventListener('click', function(event) {
        if (event.target && event.target.classList.contains('boton')) {
            alert('Botón dentro del contenedor clickeado');
        }
    });
});
```

Donde: el evento *click* se asigna al contenedor, pero solo se activa si el usuario hace clic en un botón dentro del contenedor. Esto es útil para gestionar elementos que se crean o eliminan dinámicamente.

Organización modular de JavaScript para manejo de eventos.

Para proyectos grandes, es buena práctica organizar el *JavaScript* en módulos o funciones que encapsulen diferentes partes de la lógica de eventos, manteniendo el código aún más modular y mantenible.

Ejemplo de Organización en Funciones

```
function configurarEventos() {
    const boton = document.getElementById('miBoton');

    boton.addEventListener('click', mostrarMensaje);
}

function mostrarMensaje() {
    alert('¡Hola desde una función modular!');
}

// Ejecutar la configuración de eventos al cargar la página
document.addEventListener('DOMContentLoaded', configurarEventos);
```

Donde: *configurarEventos()* organiza todos los manejadores de eventos en una sola función, que se ejecuta cuando el contenido del DOM está cargado. Esto permite reutilizar *mostrarMensaje()* en diferentes partes de la aplicación.

Ejemplo de Organización en Módulos (ES6 Modules)

Los módulos nos permiten separar la lógica en archivos distintos y cargarlos de forma modular.

```
// archivo: mensaje.js
export function mostrarMensaje() {
    alert('¡Hola desde un módulo!');
}

// archivo: main.js
import { mostrarMensaje } from './mensaje.js';

document.addEventListener('DOMContentLoaded', function () {
    const boton = document.getElementById('miBoton');
    boton.addEventListener('click', mostrarMensaje);
});
```

Donde: *main.js* carga *mostrarMensaje()* desde el módulo *mensaje.js*, separando la lógica en archivos dedicados y aumentando la claridad y reutilización del código.

 **Ejemplo:** Partimos de una página con un cuadro de texto para introducir un nombre y un botón, al pulsar sobre el botón (evento clic) si el cuadro de texto está vacío se mostrará un mensaje al usuario y se pasará el foco al cuadro de texto, si el usuario ha introducido un nombre se mostrará en la página un saludo personalizado.

```
'use strict';

// función principal para inicializar los manejadores de eventos
function inicializar() {
    const txtNombre = document.getElementById("nombre");
    const btnPulsa = document.getElementById("pulsa");
    const mainContent = document.querySelector('main');

    // función para mostrar el mensaje de saludo
    function mostrarSaludo(nombre) {
        mainContent.innerHTML += `
            <br>
            <h3>Hola ${nombre}!!!</h3>
            <p>Saludos desde JavaScript</p>
        `;
    }

    // función para manejar el evento de clic en el botón
    function manejarClic(evento) {
        evento.preventDefault(); // evita el envío del formulario si está dentro de uno

        if (txtNombre.value.trim() === "") {
            mostrarMensajeError("Debes introducir un nombre");
            txtNombre.focus();
        } else {
            mostrarSaludo(txtNombre.value);
            limpiarMensajeError();
        }
    }

    // función para mostrar un mensaje de error
    function mostrarMensajeError(mensaje) {
        let errorElemento = document.getElementById("mensajeError");
        if (!errorElemento) {
            errorElemento = document.createElement("p");
            errorElemento.id = "mensajeError";
            errorElemento.style.color = "red";
            mainContent.prepend(errorElemento);
        }
        errorElemento.textContent = mensaje;
    }
}
```

```
// función para limpiar el mensaje de error
function limpiarMensajeError() {
    const errorElemento = document.getElementById("mensajeError");
    if (errorElemento) {
        errorElemento.remove();
    }
}

// Asociar el evento de clic al botón
btnPulsa.addEventListener('click', manejarClic);
}

// Ejecutar la inicialización cuando el DOM esté cargado
document.addEventListener('DOMContentLoaded', inicializar);
```

Donde:

- *mostrarSaludo(nombre)*: función para mostrar el saludo con el nombre introducido por el usuario.
- *manejarClic(evento)*: función manejadora del evento *click* del botón.
- *mostrarMensajeError(mensaje)* y *LimpiarMensajeError()*: Para mostrar y limpiar el mensaje de error.
- *preventDefault()* para evitar la recarga de la página, incluso si el botón pulsa está en un formulario.
- *DOMContentLoaded* para ejecutar la función *inicializar* una vez que el DOM está listo, sin esperar a la carga de imágenes y otros recursos externos.

1.4 El objeto event.

El **objeto event** representa cualquier evento que ocurra en la página web, como un clic, la pulsación de una tecla, el desplazamiento, entre otros. Este objeto contiene información importante sobre el evento que se acaba de producir, lo que permite a los desarrolladores interactuar y manejar el evento de forma dinámica.

El objeto **event** se pasa automáticamente a las funciones de manejadores de eventos (*event handlers*) y proporciona **propiedades** y **métodos** que ayudan a obtener detalles del evento, como el tipo de evento, el elemento objetivo (donde se produjo el evento), la posición del ratón y más.

Normalmente, el objeto **event** se pasa como un argumento al manejador de eventos:

```
document.querySelector('button').addEventListener('click', function(evento)
{
    console.log(evento); // Muestra el objeto event en la consola
});
```

Propiedades del objeto event.

El objeto **event** proporciona propiedades útiles para obtener información del evento:

- **type**: Tipo de evento que ocurrió, como *click*, *keydown*, *submit*.
`event.type; // click`
- **target**: El elemento del DOM donde se originó el evento. Esto es útil para identificar en qué elemento se activó el evento.
`event.target; // <button id="miBoton">Haz clic</button>`
- **currentTarget**: El elemento al que se asoció el manejador de eventos. A diferencia de *target*, que se refiere al elemento original, *currentTarget* se refiere al elemento que tiene el manejador, lo cual es útil en la delegación de eventos.
`event.currentTarget; // <div id="contenedor">...</div>`
- **bubbles**: Indica si el evento se propaga (burbujea) hacia los elementos padres. Devuelve *true* si el evento está en fase de burbujeo y *false* si no.
`event.bubbles; // true o false`
- **cancelable**: Indica si el evento se puede cancelar o prevenir su acción predeterminada mediante *preventDefault()*.
`event.cancelable; // true o false`
- **defaultPrevented**: Indica si la acción predeterminada del evento ha sido prevenida (por ejemplo, si se ha llamado *preventDefault()*).
`event.defaultPrevented; // true o false`

- **timeStamp**: Marca de tiempo en milisegundos desde el inicio de la navegación hasta el momento en que ocurrió el evento.

```
event.timeStamp;
```

- **isTrusted**: Indica si el evento fue iniciado por el navegador *true* o fue creado manualmente por el desarrollador *false*.

```
event.isTrusted; // true o false
```

Métodos del objeto event.

El objeto **event** también proporciona métodos para manejar el comportamiento del evento:

- **preventDefault()**: Cancela la acción predeterminada del evento. Esto es útil, por ejemplo, para evitar que un enlace redirija a otra página o que un formulario se envíe.
- **stopPropagation()**: Detiene la propagación del evento hacia los elementos padres. Esto es útil si no deseas que el evento se maneje en otros elementos contenedores.
- **stopImmediatePropagation()**: Detiene la propagación del evento y evita que se ejecuten otros manejadores de eventos en el mismo elemento.

Propiedades específicas para eventos del ratón (MouseEvent)

Para eventos específicos como los de ratón (*click*, *mouseover*, etc.), el objeto **event** proporciona propiedades adicionales:

- **clientX** y **clientY**: Coordenadas X e Y del ratón en relación con la ventana del navegador.
- **pageX** y **pageY**: Coordenadas X e Y del ratón en relación con el documento completo.
- **screenX** y **screenY**: Coordenadas del ratón en relación con la pantalla.
- **button**: Indica qué botón del ratón se ha presionado. Los valores son: 0 (botón izquierdo), 1 (botón central), y 2 (botón derecho).
- **ctrlKey**, **shiftKey**, **altKey**, **metaKey**: Indican si alguna de las teclas especiales (Control, Shift, Alt, o Meta) estaba presionada al ocurrir el evento.

Propiedades específicas para eventos de teclado (KeyboardEvent).

En eventos del teclado como *keydown*, *keyup*, y *keypress*, el objeto **event** tiene propiedades adicionales específicas para manejar estos eventos:

- **key**: Representa la tecla presionada. Retorna una cadena que describe la tecla (Enter, Escape, a, A, etc.).
- **code**: Código físico de la tecla presionada. Representa la ubicación física de la tecla en el teclado y no cambia con la configuración del teclado.
- **repeat**: Retorna *true* si la tecla se mantiene presionada y el evento se está repitiendo.
- **ctrlKey**, **shiftKey**, **altKey**, **metaKey**: Indican si alguna de las teclas modificadoras estaba presionada.

Más información sobre el objeto event:

https://www.w3schools.com/jsref/dom_obj_event.asp
https://www.w3schools.com/jsref/obj_event.asp
https://www.w3schools.com/js/js_events.asp

Ejemplo uso de event

```
<button id="miBoton">Haz clic aquí</button>
<input type="text" id="miInput" placeholder="Escribe algo">
<p>Ver la salida por la consola del navegador</p>
<script>
    // evento de ratón
    document.getElementById('miBoton').addEventListener('click', function
    (event) {
        console.log('tipo de evento:', event.type); // click
        console.log('elemento: ', event.target); // <button id="miBoton">...
        event.preventDefault(); // prevenir la acción predeterminada (si la
        hubiera)
    });

    // evento de teclado
    document.getElementById('miInput').addEventListener('keydown', function
    (event) {
        console.log('tecla pulsada:', event.key); // tecla pulsada a, F4,..
        if (event.ctrlKey && event.key === 'z') {
            console.log('se pulso Ctrl+Z');
        }
    });
</script>
```

Ejemplo del objeto event en delegación de eventos.

En la delegación de eventos, el evento se asigna a un elemento contenedor y se utiliza el objeto **event** para identificar el elemento específico en el que ocurrió el evento.

```
document.getElementById('contenedor').addEventListener('click', function(event) {
    if (event.target.tagName === 'BUTTON') {
        console.log('Se hizo clic en un botón dentro del contenedor');
    }
});
```

Donde: **event.target** nos permite detectar el elemento en el que ocurrió el evento.

Ejemplo de creación de eventos personalizados con *CustomEvent*.

JavaScript permite la creación de eventos personalizados mediante el objeto *CustomEvent*. Estos eventos pueden llevar datos adicionales y dispararse manualmente con *dispatchEvent()*.

```
// crear un evento personalizado
let eventoPersonalizado = new CustomEvent('miEvento', {
    detail: { mensaje: 'Hola desde un evento personalizado' }
});

// asignar un manejador de eventos
document.addEventListener('miEvento', function(event) {
    console.log(event.detail.mensaje); // muestra el mensaje personalizado
});

// lanzar el evento
document.dispatchEvent(eventoPersonalizado);
```

1.5 Eventos de ratón.

Se producen cuando el usuario usa el ratón para realizar acciones sobre la interfaz. Estos eventos permiten capturar la mayoría de las interacciones posibles del usuario con el ratón en una página web.

- **click:** Se produce al pulsar con el botón izquierdo del ratón sobre un elemento.
- **dblclick:** Ocurre al hacer el usuario doble clic en un elemento.
- **contextmenu:** Se produce al pulsar el botón derecho del ratón sobre un elemento, para abrir el menú contextual de este.

- **mousemove:** Ocurre al moverse el puntero del ratón sobre un elemento.
- **mouseover:** Se produce al entrar el puntero del ratón en un elemento o en sus hijos.
- **mouseout:** Al abandonar el puntero el área de un elemento o de sus hijos.
- **mouseenter:** Similar a *mouseover*, pero no se dispara cuando el ratón entra en los elementos hijos.
- **mouseleave:** Similar a *mouseout*, pero no se dispara cuando el ratón sale de los elementos hijos.

- **mousedown:** Se produce cuando el usuario pulsa un botón del ratón sobre un elemento.
- **mouseup:** Cuando el usuario suelta un botón del ratón después de pulsarlo sobre un elemento.

- **wheel:** Ocurre cuando el usuario mueve la rueda del ratón, utilizado para detectar desplazamientos verticales o de zoom.

- **dragstart:** Se activa al comenzar a arrastrar un elemento.
- **drag:** Se dispara mientras se está arrastrando un elemento.
- **dragend:** Ocurre al soltar el elemento después de arrastrarlo.
- **dragenter:** Se dispara cuando un elemento arrastrado entra en una zona de destino.
- **dragover:** Ocurre mientras un elemento arrastrado se mantiene sobre una zona de destino.
- **dragLeave:** Se activa cuando un elemento arrastrado sale de una zona de destino.
- **drop:** Ocurre cuando un elemento arrastrado se suelta en una zona de destino.

Propiedades del objeto event en los eventos de ratón:

- **type:** Tipo de evento que ocurrió, como *click*, *mousedown*, *mousemove*, etc.
- **target:** El elemento que disparó el evento. Es el nodo en el que se hizo clic o que fue afectado por el movimiento del ratón.
- **currentTarget:** El elemento en el que el evento está siendo manejado actualmente. Esto es útil en casos de eventos delegados.
- **clientX** y **clientY:** Coordenadas del puntero del ratón en la ventana de visualización (viewport) en el momento del evento.
- **pageX** y **pageY:** Coordenadas del puntero del ratón en relación con toda la página, incluyendo el desplazamiento.
- **screenX** y **screenY:** Coordenadas del puntero del ratón en la pantalla, teniendo en cuenta la posición física en el monitor.

- **offsetX** y **offsetY**: Coordenadas del puntero del ratón en relación con el borde superior izquierdo del elemento objetivo del evento.
- **button**: Indica qué botón del ratón se presionó para iniciar el evento. Los valores comunes son:
0: Botón izquierdo
1: Botón central (rueda de desplazamiento)
2: Botón derecho
- **buttons**: Muestra un número que representa todos los botones del ratón actualmente presionados. Cada botón tiene un valor binario, que puede combinarse cuando se presionan varios botones a la vez.
- **altKey**: true si la tecla Alt estaba presionada durante el evento.
- **ctrlKey**: true si la tecla *Control* estaba presionada durante el evento.
- **shiftKey**: true si la tecla *Shift* estaba presionada durante el evento.
- **metaKey**: true si la tecla *Meta* (tecla de comando en Mac o tecla de Windows) estaba presionada durante el evento.
- **deltaX**: desplazamiento horizontal.
- **deltaY**: desplazamiento vertical.
- **deltaMode**: Un valor que indica la unidad de medida del desplazamiento (por ejemplo, 0 para píxeles, 1 para líneas).

Ejemplo acceso a propiedades en un evento de clic.

```
//javascript
document.addEventListener('click', (event) => {
    console.log('Tipo de evento:', event.type);
    console.log('Elemento objetivo:', event.target);
    console.log('Coordenadas en la ventana:', event.clientX, event.clientY);
    console.log('Coordenadas en la página:', event.pageX, event.pageY);
    console.log('Botón presionado:', event.button);
    console.log('Alt presionado:', event.altKey);
});
```

Más información eventos de ratón:

https://www.w3schools.com/jsref/obj_mouseevent.asp

1.6 Eventos de teclado.

Al producirse un evento de teclado, el objeto evento que se crea contiene propiedades específicas relacionadas con la tecla presionada.

Eventos de teclado:

- **keydown:** Al pulsar una tecla. Si la mantenemos pulsada, el evento se produce una y otra vez hasta soltarla.
- **keypress:** Al pulsar una tecla de un carácter alfanumérico (no se produce en Enter, barra espaciadora, etc.) Al mantener la tecla pulsada el evento se produce de forma continuada.
- **keyup:** Al soltar la tecla pulsada.

La secuencia de eventos al pulsar una tecla correspondiente a un carácter alfanumérico es: *keydown, keypress, keyup*. Si se mantiene pulsada la tecla, se repiten de forma continua los eventos *keydown* y *keypress*.

Cuando se pulsa otro tipo de tecla, la secuencia de eventos es: *keydown, keyup*. Si se mantiene pulsada la tecla, se repite el evento *keydown* de forma continua.

Propiedades del objeto event en eventos de teclado:

- **key:** Devuelve el valor de la tecla pulsada como una cadena (a, Enter, Escape, etc.).
- **code:** Proporciona el código físico de la tecla en el teclado, independientemente de la tecla modificadora o de la disposición del teclado (como KeyA, ArrowUp, Space, etc.).
- **keyCode:** Número que representa la tecla presionada. Se considera obsoleta y se recomienda usar *key* y *code*.
- **charCode:** Devuelve el código ASCII de la tecla presionada, pero solo en el evento *keypress*. (obsoleto).
- **altKey:** Devuelve *true* si la tecla *Alt* está pulsada.
- **ctrlKey:** Devuelve *true* si la tecla *Control* está pulsada.
- **shiftKey:** Devuelve *true* si la tecla *Shift* está pulsada.
- **metaKey:** Devuelve *true* si la tecla *Meta* (tecla de comando en Mac o tecla de Windows) está pulsada.
- **repeat:** Devuelve *true* si la tecla fue mantenida presionada, activando el evento en repetición automática.

Más información eventos de teclado:

https://www.w3schools.com/jsref/obj_keyboardevent.asp

1.7 Eventos HTML

Son eventos específicos que ocurren en el navegador, relacionados con el estado de la ventana, la carga de recursos y la interacción con elementos de formulario. Estos eventos se utilizan para manejar cambios y reacciones en el DOM, la ventana del navegador y en elementos de formularios.

Eventos HTML:

- **Load:** Ocurre en el objeto `window` cuando la página se ha cargado por completo, incluyendo todos los recursos (imágenes, scripts, etc.). También ocurre en elementos `` cuando la imagen se ha cargado completamente, y en `<object>` cuando el objeto se ha cargado.
- **unLoad:** Se dispara en el objeto `window` cuando la página se está cerrando o recargando. También ocurre en elementos `<object>` cuando el objeto se elimina de la página.
- **abort:** Ocurre cuando el usuario detiene la descarga de un elemento antes de que termine, como en un `<object>` que no ha terminado de cargarse.
- **error:** Se dispara en el objeto `window` cuando ocurre un error en el código JavaScript. También se produce en `` cuando una imagen no puede cargarse completamente, o en `<object>` cuando el objeto no se carga correctamente.
- **select:** Ocurre cuando el usuario selecciona texto en los elementos `<input>` o `<textarea>`.
- **change:** Ocurre en `<input>` y `<textarea>` cuando el elemento pierde el foco y su valor ha cambiado. En el elemento `<select>`, se dispara cada vez que el usuario selecciona una opción diferente.
- **submit:** Ocurre cuando se hace clic en un botón de tipo `submit` en un formulario, lo que intenta enviar el formulario.
- **reset:** Se produce cuando se hace clic en un botón de tipo `reset`, lo cual restablece todos los campos de un formulario a sus valores predeterminados.
- **resize:** Ocurre en el objeto `window` cuando la ventana del navegador es redimensionada, permitiendo ajustar el contenido en respuesta al nuevo tamaño.
- **scroll:** Ocurre cuando el usuario desplaza (hace scroll) un elemento que tiene barra de desplazamiento, como una ventana o un contenedor con `overflow`.
- **focus:** Se produce cuando un elemento, como un campo de entrada, obtiene el foco. Útil para ejecutar funciones al entrar en un campo específico.
- **blur:** Ocurre cuando un elemento pierde el foco. Por ejemplo, al salir de un campo de entrada.

Más información eventos html: https://www.w3schools.com/jsref/dom_obj_event.asp

1.8 Eventos del DOM

Los eventos del DOM relacionados con la modificación de la estructura del árbol DOM pertenecen a la especificación **DOM Level 3**. Se producen cuando cambia el árbol DOM.

Eventos del DOM:

- ***DOMSubtreeModified***: Se produce al modificar el árbol de un elemento o documento (añadir, eliminar o modificar).
- ***DOMNodeInserted***: Se produce cuando se añade un nodo hijo a un nodo padre. Se produce tanto para el nodo padre como para el nodo recién añadido.
- ***DOMNodeRemoved***: Se produce cuando se elimina un nodo hijo de un nodo padre. Este evento se dispara tanto para el nodo padre como para el nodo eliminado.
- ***DOMNodeRemovedFromDocument***: Se produce cuando un nodo se elimina del documento por completo. Esto significa que el nodo se ha eliminado del árbol DOM y ya no es accesible en el documento.
- ***DOMNodeRemovedFromDocument***: Se produce cuando un nodo se elimina del documento por completo. Esto significa que el nodo se ha eliminado del árbol DOM y ya no es accesible en el documento.
- ***DOMNodeInsertedIntoDocument***: Se produce cuando un nodo se inserta en el documento, es decir, cuando pasa a formar parte del árbol DOM principal.

La mayoría de estos métodos están obsoletos.

2 Utilización de formularios.

En HTML, los formularios permiten capturar y enviar datos introducidos por el usuario hacia un servidor o manejarlos en el lado del cliente mediante JavaScript. En el modelo de objetos del documento (DOM), los formularios son representados como elementos en un array accesible a través de `document.forms`, lo que permite manipularlos directamente a través de JavaScript.

Estructura de un Formulario en HTML

Un formulario se representa mediante la etiqueta `<form>`, que contiene los diferentes campos y controles que permiten al usuario introducir y seleccionar información. El formulario se representa internamente como un objeto JavaScript `HTMLFormElement`, que hereda del objeto `HTMLElement` todas sus propiedades y métodos, además de otros para la gestión de formularios.

Componentes de un Formulario

Un formulario se compone esencialmente de las siguientes partes:

1. **Etiqueta `<form>`:** Es el contenedor que delimita el formulario. Dentro de este elemento se colocan todos los controles y elementos de entrada. La etiqueta `<form>` puede incluir atributos importantes que controlan cómo y dónde se envían los datos.
2. **Controles de Formulario:** Son los elementos interactivos donde el usuario puede introducir o seleccionar información. Estos incluyen campos de entrada (`<input>`), áreas de texto (`<textarea>`), menús de selección (`<select>`), casillas de verificación (`<input type="checkbox">`), botones de opción (`<input type="radio">`), entre otros.
3. **Etiquetas de Identificación:** Las etiquetas (`<label>`) ayudan a identificar los controles del formulario, mejorando la accesibilidad y la experiencia del usuario. Cada etiqueta se asocia con un control específico mediante el atributo `for` o, al envolver el control, haciendo que sea seleccionable con un clic.
4. **Botón de Envío:** Generalmente, el formulario incluye un botón (`<button type="submit">` o `<input type="submit">`) que dispara el envío de los datos cuando el usuario termina de completar el formulario. También se pueden incluir botones de tipo `reset` para limpiar los campos del formulario.

Atributos del Elemento `<form>`.

El elemento `<form>` permite especificar varios atributos clave que definen el comportamiento del formulario:

- **action:** Especifica la URL a la que se enviarán los datos cuando se haga clic en el botón de envío. Si este atributo se omite, los datos se envían a la misma página donde se encuentra el formulario.

- **method:** Determina el método HTTP que se utilizará para enviar los datos. Los métodos más comunes son:
 - **GET:** Envía los datos como parte de la URL, en la cadena de consulta (*?nombre=valor&nombre2=valor2*). Se usa para solicitudes de solo lectura y se quiere que los datos enviados aparezcan en la URL.
 - **POST:** Envía los datos en el cuerpo de la solicitud HTTP, protegiendo los datos de exposición en la URL. Este método es adecuado para envíos de datos grandes o sensibles.
- **enctype:** Define el tipo de codificación de los datos que se envían. Es relevante solo para el método **POST**. Los valores más comunes son:
 - **application/x-www-form-urlencoded** (por defecto): Codifica los datos en pares de *nombre=valor*, separados por &.
 - **multipart/form-data**: Necesario cuando se envían archivos. Permite enviar datos binarios, como imágenes.
 - **text/plain**: Envía los datos en texto sin formato, sin ningún tipo de codificación, aunque raramente se usa.
- **target:** Define dónde se abrirá el recurso especificado en el *action*. Puede ser:
 - **_self** (por defecto): Carga la respuesta en el mismo marco de la ventana.
 - **_blank**: Abre la respuesta en una nueva ventana o pestaña.
 - **_parent**: Carga la respuesta en el marco principal si el documento está dentro de un marco.
 - **_top**: Carga la respuesta en la ventana completa.

Ejemplo de Estructura de un Formulario:

```
<form action="https://ejemplo.es/submit" method="POST" enctype="multipart/form-data">
  <label for="nombre">Nombre:</label>
  <input type="text" id="nombre" name="nombre" required>

  <label for="email">Correo electrónico:</label>
  <input type="email" id="email" name="email" required>

  <label for="mensaje">Mensaje:</label>
  <textarea id="mensaje" name="mensaje" rows="4" cols="50"></textarea>

  <label for="archivo">Adjuntar archivo:</label>
  <input type="file" id="archivo" name="archivo">

  <button type="submit">Enviar</button>
  <button type="reset">Restablecer</button>
</form>
```

Propiedades y Métodos del Objeto `HTMLFormElement`

El objeto `HTMLFormElement` permite acceder y manipular los datos y comportamientos del formulario a través de JavaScript.

Propiedades:

- **elements:** Devuelve una colección de todos los elementos de formulario (controles) dentro del formulario. Cada elemento se puede acceder por su `name` o índice.
- **length:** Indica la cantidad de controles dentro del formulario.
- **action:** Devuelve o establece la URL de destino a la que se enviarán los datos.
- **method:** Devuelve o establece el método de envío (`GET`, `POST`, etc.).
- **target:** Devuelve o establece el destino del envío (`_self`, `_blank`, etc.).
- **name:** Especifica o devuelve el nombre del formulario, lo cual permite referenciarlo fácilmente en JavaScript.

Métodos:

- **submit():** Envía el formulario. Es útil para enviar el formulario programáticamente sin depender del botón de envío.
- **reset():** Restablece todos los valores de los controles del formulario a sus valores predeterminados.
- **checkValidity():** Verifica la validez de los elementos del formulario que tienen restricciones de validación (`required`, `pattern`, etc.). Devuelve `true` si todos los elementos son válidos.
- **reportValidity():** Similar a `checkValidity()`, pero muestra un mensaje de advertencia si un elemento no es válido.

Validación mediante atributos html:

La validación de los formularios asegura que el usuario ingrese datos correctos antes de enviar la información. HTML5 permite realizar validaciones usando atributos como:

- **required:** Obliga al usuario a completar el campo.
- **pattern:** Define una expresión regular que el valor debe cumplir.
- **min** y **max:** Define el valor mínimo y máximo para campos numéricos.
- **minlength** y **maxlength:** Define la longitud mínima y máxima de texto permitida en campos de texto.

Más información sobre formularios:

http://www.w3schools.com/html/html_form_input_types.asp

<https://developer.mozilla.org/es/docs/HTML/Element/form>

http://www.w3schools.com/html/html_forms.asp

https://www.w3schools.com/html/html5_new_elements.asp

2.1 Utilización de formularios desde JavaScript.

Los formularios permiten interactuar de manera dinámica con los datos de introducidos por los usuarios. Con JavaScript, podemos controlar el envío de formularios, validar datos, gestionar errores y realizar acciones basadas en la entrada del usuario, todo sin recargar la página.

Acceso a Elementos del Formulario

Para interactuar con un formulario en JavaScript, es común seleccionar el formulario y sus elementos de entrada en el DOM. Podemos acceder al formulario y sus elementos de varias formas:

- **Por ID:** Accede al formulario directamente usando su id.
`let formulario = document.getElementById('miFormulario');`
- **Por el índice en `document.forms`:** Si tenemos varios formularios, podemos acceder a ellos mediante `document.forms`.
`let formulario = document.forms[0]; //primer formulario en el DOM`
- **Por elementos dentro del formulario:** Podemos acceder a los elementos individuales mediante su `name` o `id`.
`let nombre = document.getElementById('nombre');`
`let email = formulario.elements['email'];`

Captura del Evento submit

Uno de los eventos más importantes en un formulario es `submit`, que se dispara cuando se intenta enviar el formulario. Para evitar que el formulario se envíe y recargue la página de inmediato, podemos usar `preventDefault()` para manejar los datos en el mismo cliente.

```
<form id="miFormulario">
  <input type="text" id="nombre" name="nombre" placeholder="Tu nombre">
  <input type="email" id="email" name="email" placeholder="Tu email">
  <button type="submit">Enviar</button>
</form>

<script>
  let formulario = document.getElementById('miFormulario');
  formulario.addEventListener('submit', function(event) {
    event.preventDefault(); // evitar el envío del formulario
    console.log('validando datos...');

  });
</script>
```

Donde: el evento `submit` se captura y se previene el comportamiento predeterminado para que el formulario no se envíe hasta que se realicen las validaciones necesarias.

3 Validación y envío de formularios.

La validación de formularios es fundamental para asegurarse de que los datos proporcionados por los usuarios son correctos y cumplen con los requisitos. Existen dos tipos de validación: **validación en el lado del cliente** (con HTML o JavaScript) y **validación en el lado del servidor** (para una seguridad completa):

- **Validaciones en el Servidor:** Las validaciones en el servidor y la base de datos son esenciales para asegurar la integridad y seguridad de la información que llega al sistema. Estas validaciones ocurren cuando el servidor recibe una petición, la procesa, y emite una respuesta. Si el usuario incluye datos incorrectos, el servidor puede sobrecargarse al procesar estas solicitudes defectuosas. Para reducir esta carga, realizamos una primera capa de filtrado en el cliente, evitando que datos erróneos lleguen al servidor innecesariamente.

- **Validaciones en el Cliente:** Estas validaciones las realizamos en el navegador mediante JavaScript. Al pulsar el botón de envío de formulario, los scripts en el cliente comprueban los datos introducidos por el usuario y verifican si cumplen con los requisitos necesarios. Si algún dato es incorrecto o falta, se muestra un mensaje al usuario indicando las correcciones necesarias, evitando el envío de datos inválidos al servidor. Solo cuando los datos son validados como correctos se envía la solicitud al servidor, optimizando el proceso y reduciendo la carga de trabajo en el servidor.

Para validar y enviar formularios en JavaScript, debemos:

1. Capturar el evento de envío del formulario.
2. Validar los campos de entrada para asegurarnos de que cumplen con los requisitos especificados (campos obligatorios, longitud mínima, coincidencia de patrones, etc.).
3. Mostrar mensajes de error si algún campo no es válido.
4. Envía el formulario usando JavaScript.

Para implementar validaciones efectivas en formularios web, debemos estructurarlos de forma que, **al detectar un intento de envío, se active una función de validación**. Esta función analizará los datos proporcionados por el usuario y verificará que cada campo cumpla con las restricciones establecidas, tales como la obligatoriedad de ciertos campos, el formato adecuado de datos y otras reglas de validación específicas.

Para asegurar que el formulario se valide correctamente antes de enviar los datos al servidor, es fundamental utilizar el **evento submit**. Este evento permite interceptar el proceso de envío y ejecutar las comprobaciones necesarias, deteniendo el envío si algún campo no cumple con los requisitos especificados.

Ejemplo:

```

<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Formulario con Validación</title>
</head>
<body>
    <form id="formulario">
        <label for="nombre">Nombre:</label>
        <input type="text" id="nombre" name="nombre" required>
        <small id="nombreError" class="error"></small>

        <label for="email">Email:</label>
        <input type="email" id="email" name="email" required>
        <small id="emailError" class="error"></small>

        <label for="password">Contraseña:</label>
        <input type="password" id="password" name="password" required minlength="6">
        <small id="passwordError" class="error"></small>

        <button type="submit">Enviar</button>
    </form>

    <script src="script.js"></script>
</body>
</html>

// script.js
document.getElementById("formulario").addEventListener("submit", function (event)
{
    event.preventDefault(); // evita el envío automático del formulario

    // limpiar mensajes de error previos
    limpiarErrores();

    // obtener los valores de los campos
    const nombre = document.getElementById("nombre").value;
    const email = document.getElementById("email").value;
    const password = document.getElementById("password").value;

    // validar campos
    let esValido = true;

    if (nombre === "") {
        mostrarError("nombreError", "El nombre es obligatorio.");
        esValido = false;
    }
}

```

```

if (!validarEmail(email)) {
    mostrarError("emailError", "El email no es válido.");
    esValido = false;
}

if (password.length < 6) {
    mostrarError("passwordError", "La contraseña debe tener al menos 6
caracteres.");
    esValido = false;
}

// Si todos los campos son válidos, enviar el formulario
if (esValido) {
    alert("Formulario enviado con éxito");
    // enviaremos el formulario
}
});

function mostrarError(id, mensaje) {
    const errorElement = document.getElementById(id);
    errorElement.textContent = mensaje;
    errorElement.style.color = "red";
}

function limpiarErrores() {
    document.querySelectorAll(".error").forEach(error => {
        error.textContent = "";
    });
}

function validarEmail(email) {
    const regex = /^[^@\s]+@[^\s@]+\.\w+$/;
    return regex.test(email);
}

```

¿Qué debemos validar en un formulario?

- Comprobar que se han introducido todos los campos obligatorios.
- Comprobar que el formato de los campos es el esperado (fechas, NIF/NIE, teléfonos, etc.).
- Comprobar la validez de direcciones de correo y URLs.
- Comprobar que no se supera la longitud, número de líneas o tamaño de la entrada.

█ Ejemplo: formulario con los campos nombre, contraseña y repetir contraseña, validar que los tres campos son obligatorios y que el campo contraseña y el campo repetir contraseña son iguales.

```
// función de inicialización al cargar la página
function inicializar() {
    const nombre = document.getElementById('nombre');
    const pass = document.getElementById('pass');
    const repass = document.getElementById('repass');
    const btnEnviar = document.getElementById('enviar');
    const formulario = document.querySelector('form');
    const mainContent = document.querySelector('main');

    // asociar el evento click al botón de enviar
    btnEnviar.addEventListener('click', (e) => {
        e.preventDefault(); // evitar el envío del formulario

        // validar los campos de nombre, contraseña y repetición de contraseña
        const nombreValido = validaObligatorio(nombre, "Debe introducir un nombre");
        const passValido = validaObligatorio(pass, "Debe introducir una contraseña");
        const repassValido = validaObligatorio(repass, "Debe repetir la contraseña");
        const contrasenasCoincidan = repetirContrasena(pass, repass);

        if (nombreValido && passValido && repassValido && contrasenasCoincidan) {
            mostrarMensajeExito("Los datos son correctos, el formulario se enviará");
            formulario.submit();
        }
    });
}

// función para validar campo obligatorio y mostrar mensaje de error
function validaObligatorio(campo, mensajeError) {
    if (campo.value.trim() === "") {
        mostrarMensajeError(campo, mensajeError);
        campo.focus();
        return false;
    }
    limpiarMensajeError(campo);
    return true;
}

// función para comprobar que las contraseñas coincidan
function repetirContrasena(campo1, campo2) {
    if (campo1.value !== campo2.value) {
        mostrarMensajeError(campo2, "Las contraseñas introducidas no coinciden");
        campo2.value = "";
        campo2.focus();
        return false;
    }
    limpiarMensajeError(campo2);
    return true;
}
```

```

}

// función para mostrar mensaje de error asociado al campo
function mostrarMensajeError(campo, mensaje) {
    let errorElemento = campo.nextElementSibling;
    if (!errorElemento || !errorElemento.classList.contains('mensaje-error')) {
        errorElemento = document.createElement("span");
        errorElemento.classList.add("mensaje-error");
        errorElemento.style.color = "red";
        campo.insertAdjacentElement('afterend', errorElemento);
    }
    errorElemento.textContent = mensaje;
}

// función para limpiar mensaje de error del campo
function limpiarMensajeError(campo) {
    const errorElemento = campo.nextElementSibling;
    if (errorElemento && errorElemento.classList.contains('mensaje-error')) {
        errorElemento.remove();
    }
}

// función para mostrar un mensaje de éxito al usuario
function mostrarMensajeExito(mensaje) {
    let mensajeExito = document.getElementById('mensajeExito');
    if (!mensajeExito) {
        mensajeExito = document.createElement("p");
        mensajeExito.id = "mensajeExito";
        mensajeExito.style.color = "green";
        mainContent.prepend(mensajeExito);
    }
    mensajeExito.textContent = mensaje;
}

// ejecutar la inicialización cuando el DOM esté cargado
document.addEventListener('DOMContentLoaded', inicializar);

```

Buenas Prácticas para Validación y Envío

- Validar en el cliente y el servidor: Aunque JavaScript puede manejar la validación en el cliente, nunca depende solo de él por cuestiones de seguridad.
- Usar mensajes de error claros: Mostrar al usuario mensajes de error comprensibles y específicos para mejorar la experiencia de usuario.
- Implementar validación en tiempo real: Para ciertos formularios, el evento *input* permite mostrar errores inmediatamente mientras el usuario escribe.
- Aplicar CSS para errores: Cambiar el estilo de campos con errores (por ejemplo, borde rojo) mejora la visualización.

3.1 Algunos ejemplos de funciones de validación de datos

Validación de campo obligatorio

```
// ----- función que comprueba los campos obligatorios -----
function compruebaObligatorio(campo) {
    let error = false;
    // verifica si el valor del campo está vacío o es nulo
    if (campo.value.trim() === "") {
        // muestra un mensaje de alerta
        alert("El campo '" + campo.name.replace("_", " ") + "' es obligatorio.");

        campo.focus();
        error = true;
    }
    return error;
}
```

Validación de campos numéricos

```
// ----- función que comprueba los campos numéricos -----
function compruebaNumerico(campo) {
    let error = false;

    // Convierte el valor del campo a un número para una comparación precisa
    const valorNumerico = Number(campo.value);

    // Verifica si el valor es numérico y positivo
    if (isNaN(valorNumerico) || valorNumerico <= 0) {
        alert("El campo '" + campo.name + "' debe ser un número mayor que cero.");

        // pasa el foco al campo
        campo.focus();

        error = true;
    }

    return error;
}

// ----- función que comprueba los campos numéricos -----
function compruebaNumerico(campo) {
    const valorNumerico = Number(campo.value);
    // que el valor no sea NaN y sea un número positivo
    return !isNaN(valorNumerico) && valorNumerico > 0;
}
```

Validar Checkbox marcado

```
//----- validar checkbox marcado -----
function comprobarCheckbox(campo) {
    return campo.checked;
}
```

Validar código postal

```
//----- código postal -----
function compruebaCodigoPostal(campo) {
    // si el valor del campo es numérico y tiene 5 dígitos
    const esNumerico = !isNaN(campo.value) && campo.value.trim() !== "";
    const longitudCorrecta = campo.value.length === 5;

    // devuelve true si hay un error en el código postal
    return !(esNumerico && longitudCorrecta);
}
```

Validar repetir dirección de correo.

```
// ----- repetir email -----
function repetirEmail(campo, campo1) {
    return campo.value === campo1.value;
}
```

4 Expresiones regulares.

Las expresiones regulares son patrones que permiten buscar, validar, o manipular cadenas de texto de manera eficiente. En JavaScript y otros lenguajes de programación, se utilizan comúnmente para verificar que un texto cumple con un formato específico (como correos electrónicos, números de teléfono, direcciones IP, etc.) o para realizar búsquedas y reemplazos en textos.

Sintaxis Básica de Expresiones Regulares.

En JavaScript, las expresiones regulares se definen entre barras diagonales `/.../` o usando el constructor `new RegExp()`.

Los principales elementos y constructores de una expresión regular son:

Metacaracteres básicos.

- `.` : Representa cualquier carácter excepto una nueva línea. Ejemplo: `/a.c/` encuentra *abc*, *a1c*, *a_c*, etc.
- `^` : Indica el inicio de una línea. Ejemplo: `/^abc/` busca *abc* solo al inicio de la cadena.
- `$` : Indica el final de una línea. Ejemplo: `/abc$/` busca *abc* solo al final de la cadena.
- `\d` : Representa cualquier dígito (0-9). Equivalente a `[0-9]`.
- `\D` : Representa cualquier carácter que no sea un dígito.
- `\w` : Representa cualquier carácter alfanumérico o guion bajo (`[A-Za-z0-9_]`).
- `\W` : Representa cualquier carácter que no sea alfanumérico.
- `\s` : Representa cualquier carácter de espacio (espacio, tabulador, salto de línea).
- `\S` : Representa cualquier carácter que no sea un espacio en blanco.

Cuantificadores.

- `*` : Coincide con cero o más repeticiones del carácter anterior. Ejemplo: `/a*/` encuentra *a*, *aa*, o ninguna *a*.
- `+` : Coincide con una o más repeticiones del carácter anterior. Ejemplo: `/a+/` encuentra *a*, *aa*, pero no ninguna *a*.
- `?` : Coincide con cero o una repetición del carácter anterior. Ejemplo: `/a?` encuentra una *a* o ninguna.

- **{n}** : Coincide exactamente con *n* repeticiones del carácter anterior. Ejemplo: */a{3}/* encuentra *aaa*.
- **{n,}** : Coincide con *n* o más repeticiones. Ejemplo: */a{2, }/* encuentra *aa*, *aaa*, etc.
- **{n,m}** : Coincide con al menos *n* y como máximo *m* repeticiones. Ejemplo: */a{2,4}/* encuentra *aa*, *aaa* o *aaaa*.

Agrupamiento y alternancia:

- **(...)** : Agrupa caracteres o expresiones. Ejemplo: */(abc)+/* busca una o más repeticiones de *abc*.
- **|** : Representa una opción entre varias. Ejemplo: */a/b/* encuentra *a* o *b*.

Clases de caracteres y rangos:

- **[abc]** : Coincide con cualquiera de los caracteres en el conjunto. Ejemplo: */[aeiou]/* busca cualquier vocal.
- **[^abc]** : Coincide con cualquier carácter que no esté en el conjunto. Ejemplo: */[^aeiou]/* busca cualquier carácter que no sea una vocal.
- **[a-z]** : Coincide con cualquier carácter dentro del rango especificado. Ejemplo: */[a-z]/* encuentra cualquier letra minúscula.
- **[0-9]** : Coincide con cualquier dígito entre 0 y 9.

Modificadores (Flags):

- **g** : Busca todas las coincidencias en lugar de solo la primera. Ejemplo: */a/g*.
- **i** : Hace que la búsqueda no distinga entre mayúsculas y minúsculas. Ejemplo: */a/i* encuentra *a* y *A*.
- **m** : Habilita el modo de varias líneas. **^** y **\$** funcionan en cada línea en lugar de solo al inicio y al final de toda la cadena.

4.1 Métodos en JavaScript para trabajar con expresiones regulares.

JavaScript proporciona varios métodos para trabajar con expresiones regulares:

- **test()**: Verifica si una expresión regular coincide en una cadena.

```
const regex = /\d+/  
regex.test("123"); // true
```

- **match()**: Busca coincidencias y devuelve un array con todas las coincidencias o *null* si no encuentra ninguna.

```
const texto = "Hola 123";  
texto.match(/\d+/g); // ["123"]
```

- **replace()**: Reemplaza coincidencias en la cadena por un nuevo valor.

```
const texto = "Precio: 1500";  
texto.replace(/\d+/, "2000"); // "Precio: 2000"
```

- **split()**: Divide una cadena en un array de subcadenas, utilizando la expresión regular como delimitador.

```
const texto = "manzana,pera,plátano";  
texto.split(/,/); // ["manzana", "pera", "plátano"]
```

Más información expresiones regulares:

http://www.w3schools.com/jsref/jsref_obj_regexp.asp

http://es.wikipedia.org/wiki/Expresi%C3%B3n_regular

https://developer.mozilla.org/es/docs/Gu%C3%A1a_de_JavaScript_1.5/Objetos_base_preditados/Objeto_RegExp

4.2 Validar un formulario con expresiones regulares.

Con la combinación de estas expresiones podemos realizar patrones para validar datos en los formularios. Combinando cada una de las condiciones obtenemos patrones como pueden ser, *una dirección de correo electrónico, un número de teléfono, un código postal, un DNI, etc.*

 Ejemplo formulario con dos campos: *nombre de usuario* y *correo electrónico* y un botón para el envío. Se realizarán las siguientes validaciones: el campo *nombre* debe tener entre 2 y 15 caracteres permitiendo letras, números y espacios en blanco y el campo *Email*, con formato correcto de correo electrónico.

```
'use strict';

// al cargar la página
window.addEventListener('load', function () {
    const mensajes = [
        'El nombre debe tener entre 2 y 20 caracteres alfanuméricos',
        'Debe introducir una dirección de correo válida',
        'Datos correctos. El formulario se enviará'
    ];

    const nombre = document.getElementById('nombre'); // campo nombre
    const email = document.getElementById('email'); // campo email
    const btnEnviar = document.getElementById('enviar'); // botón enviar

    btnEnviar.addEventListener('click', (evento) => {
        evento.preventDefault(); // evita el envío del formulario por defecto

        // validaciones
        if (!validaNombre(nombre)) {
            nombre.focus();
            mostrarMensaje(mensajes[0]);
        } else if (!validaEmail(email)) {
            email.focus();
            mostrarMensaje(mensajes[1]);
        } else {
            mostrarMensaje(mensajes[2]);
            document.querySelector('form').submit();
        }
    });
});

// función que valida un nombre
function validaNombre(campo) {
    const nomexpreg = /^[a-zA-Z0-9\s]{2,20}$/; // Entre 2 y 20 caracteres
    alfanuméricos, permite espacios
    return nomexpreg.test(campo.value.trim());
}
```

```
// función que valida el formato de correo
function validaEmail(campo) {
    const emailexprg = /^[\\w.-]+@[\\w-]+\\.+[a-zA-Z]{2,4}$/; // formato de email
    return emailexprg.test(campo.value.trim());
}

// función que muestra los mensajes
function mostrarMensaje(mensaje) {
    alert(mensaje);
}
```

Algunos ejemplos de funciones de validación con expresiones regulares.

Validar un NIF.

Un dato habitual en los formularios es el Número de Identificación Fiscal (NIF).

```
//-----      nif      -----
function validaNIF(nif) {

    const regex = /^\\d{8}[A-Za-z]$/;
    let esCorrecto = false;

    if (regex.test(nif)) {
        // separamos el número y la letra del NIF
        const numero = parseInt(nif.substring(0, 8));
        const letra = nif.charAt(8).toUpperCase();

        // calculamos la letra correcta
        const letrasValidas = "TRWAGMYFPDXBNJZSQVHLCKE";
        const letraCalculada = letrasValidas.charAt(numero % 23);

        // comparamos la letra del NIF con la letra calculada
        esCorrecto = letra === letraCalculada;
    }

    return esCorrecto;
}
```

Validar un número de teléfono.

Debemos asegurarnos de que el número de teléfono tiene un formato correcto.

```
//----- teléfono/móvil -----
function validaTelefono(telefono){
    return (/^([9|6|7]{1})[0-9]{8}$/.test(telefono))
}

//----- teléfono -----
function compruebaTelefono(campo) {
    var error = false;
    if (!(/^([9]{1})[0-9]{8}$/.test(campo.value))) {
        alert("El formato del 'teléfono' no es correcto");
        campo.focus();
        error = true;
    }
    return error;
}

//----- móvil -----
function compruebaMovil(campo) {
    var error = false;
    if (!(/^([6|7]{1})[0-9]{8}$/.test(campo.value))) {
        alert("El formato del 'móvil' no es correcto");
        campo.focus();
        error= true;
    }
    return error;
}
```

Validar un código postal.

```
//----- código postal -----
function compruebaCodigoPostal(campo) {
    let error = false;
    if (!(/^\d{5}$/.test(campo.value.trim()))) {
        alert("El formato del código postal no es correcto");
        campo.focus();
        error = true;
    }
    return error;
}

//----- código postal -----
function compruebaCodigoPostal(campo) {
    return ((!isNaN(campo)) && (campo.length !=5));
}
```

```
//----- código postal -----
function compruebaCodigoPostal(campo) {
    return (/^[\d]{5}$/.test(campo.value));
}
```

Validar campo sólo letras.

```
// ----- función que comprueba sólo letras -----
function validaSoloLetras(campo){
    return (/^[a-zA-Z]+$/ .test(campo.value))
}
```

Validar campo sólo letras, minúsculas acentuadas y espacios.

```
// ----- función que comprueba los campos alfabéticos -----
function compruebaCaracteresEspacio(campo) {
    const regex = /^[a-zA-Z áéíóúÁÉÍÓÚ]+$/;

    if (!regex.test(campo.value.trim())) {
        mostrarMensajeError(`El campo '${campo.name}' solo permite letras y
espacios.`);
        campo.focus();
        return true;
    }

    return false;
}

function mostrarMensajeError(mensaje) {
    // aquí podemos personalizar cómo mostrar los mensajes de error
    // por ejemplo, mostrando el mensaje en un contenedor específico en la página
    alert(mensaje);
}
```

- Hay otros datos que podemos validar con expresiones regulares, como: *números de cuentas bancarias, CIF de empresas, teléfonos internacionales* y cualquier otro dato que siga un patrón específico.
- Validar los datos para que el usuario no envíe el formulario si estos no son correctos aumenta el rendimiento del servidor, al disminuir las peticiones. Esto contribuye a un rendimiento más eficiente del sistema y a una experiencia de usuario más fluida.
- También es importante la claridad en los mensajes de validación para una buena usabilidad. Es esencial que los mensajes de error indiquen de manera específica cómo el usuario debe introducir los datos, para evitar confusión o frustración. Por ejemplo, en un campo de teléfono, un mensaje como “Por favor, introduce un número de 9 dígitos” es mucho más útil que un mensaje genérico como “Formato incorrecto”. Además, es recomendable incluir ejemplos o indicaciones en cada campo, como "Formato: 12345" para códigos postales, lo cual reduce errores y mejora la experiencia del usuario.
- Ser demasiado estricto en las validaciones puede tener efectos negativos, especialmente para usuarios que no estén familiarizados con la aplicación. Validaciones excesivas o restrictivas pueden hacer que la curva de aprendizaje de la aplicación sea demasiado alta, llevando a la frustración y al abandono de la página.
- En los casos de validaciones estrictas, es importante contar con un sistema de soporte accesible y claro para ayudar al usuario en caso de dudas. Dependiendo de la infraestructura de la aplicación, este soporte puede ser proporcionado mediante un servicio de atención al cliente telefónico, asistencia en línea, o correo electrónico. Además, se podemos integrar un apartado de ayuda o una sección de preguntas frecuentes (FAQ) en la misma aplicación para guiar al usuario en caso de errores recurrentes.