Formularios, eventos y dom

Contenido

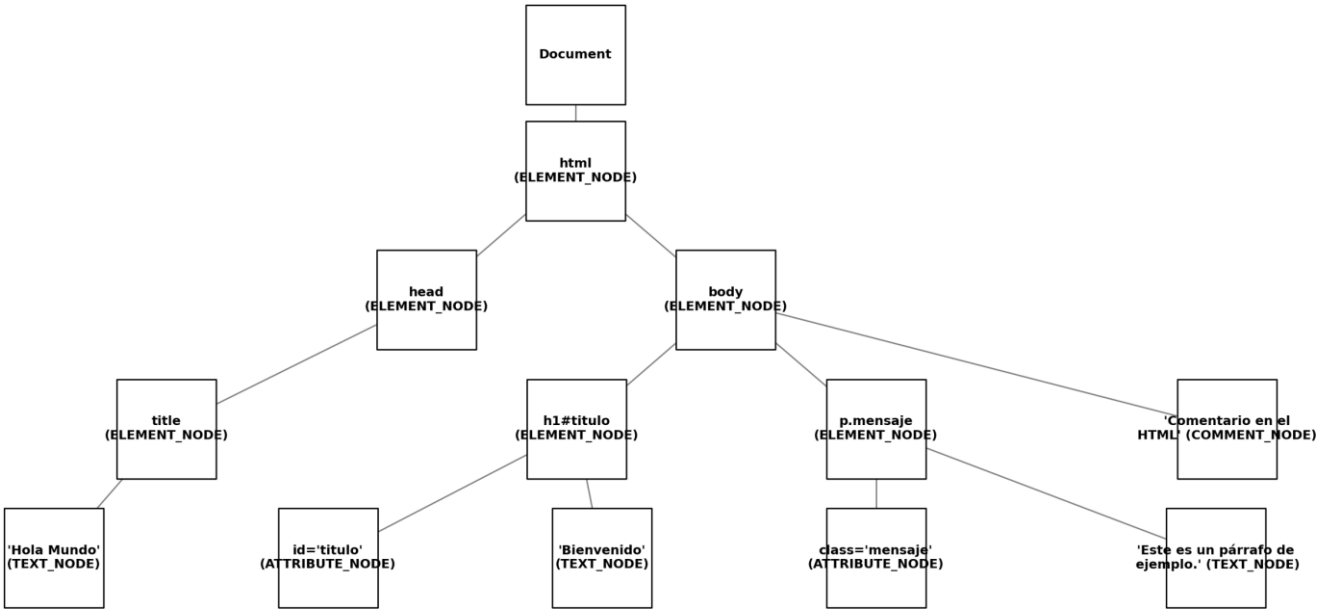
El DOM

El **Document Object Model (DOM)** es una interfaz de programación que los navegadores utilizan para representar y manipular el contenido de una página web. Cuando se carga una página en el navegador, este interpreta el código HTML y lo convierte en una estructura jerárquica conocida como el árbol DOM. Esta estructura permite a los desarrolladores acceder y modificar elementos, atributos, y contenido de la página de manera dinámica utilizando JavaScript.

Por ejemplo

```
<!DOCTYPE html>
<html>
<head>
    <title>Hola Mundo</title>
</head>
<body>
    <h1 id="titulo">Bienvenido</h1>
    <p class="mensaje">Este es un párrafo de ejemplo.</p>
    <!-- Comentario en el HTML -->
</body>
</html>
```

Genera la siguiente estructura jerárquica de nodos.

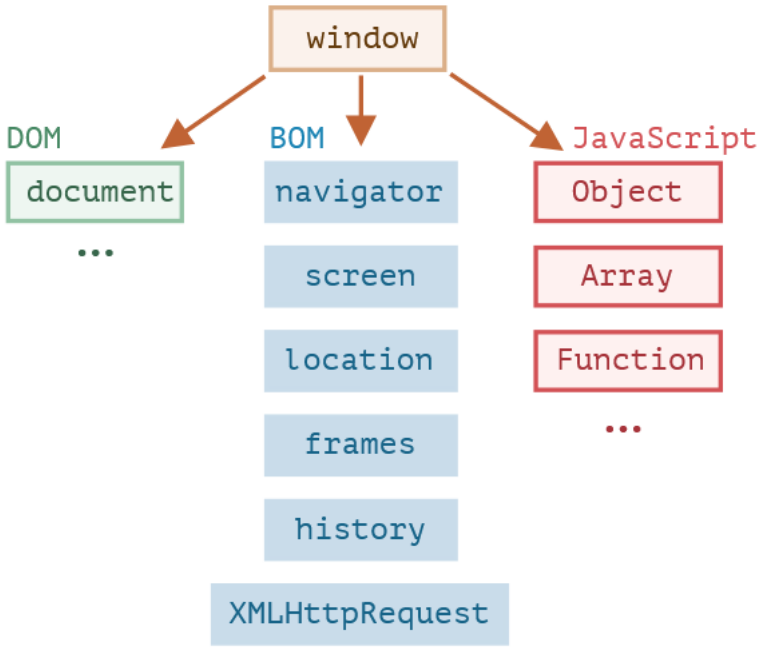


Para trabajar con el DOM contamos con dos tipos de objetos principales:

- **window**: Representa la ventana del navegador y es el objeto donde se definen las variables globales (***this*** y ***globalThis*** según el caso apuntan a ***window***).
- **document**: Representa el documento HTML que se carga en la ventana.

Adicionalmente disponemos del **Browser Object Model (BOM)** que proporciona un conjunto de objetos que permiten interactuar con la ventana del navegador y el entorno de la página web. A diferencia del **DOM**, que se centra en la estructura del documento HTML, el BOM proporciona acceso a elementos específicos del navegador, permitiendo controlar la ventana, la URL, el historial y más.

- **navigator:** Proporciona información sobre el navegador del usuario, como su nombre, versión, sistema operativo y soporte de ciertas características (como ***navigator.geolocation***).
- **screen:** Ofrece información sobre la pantalla del usuario, incluyendo la resolución (***screen.width, screen.height***) y el espacio disponible (***screen.availWidth, screen.availHeight***).
- **location:** Contiene información sobre la URL actual de la página y permite redirigir al usuario a otra URL (***location.href, location.reload(), location.replace()***).
- **history:** Permite interactuar con el historial de navegación del usuario. Proporciona métodos para moverse hacia atrás y hacia adelante (***history.back(), history.forward(), history.go()***).
- **localStorage:** Proporciona almacenamiento persistente en el navegador. Los datos guardados permanecen incluso después de que el usuario cierre el navegador (***localStorage.setItem(), localStorage.getItem()***).
- **sessionStorage:** Similar a ***localStorage***, pero los datos se eliminan al cerrar la ventana o pestaña del navegador. Se usa para almacenamiento temporal por sesión.
- **console:** Utilizado para depuración. Permite mostrar mensajes en la consola del navegador (***console.log(), console.error(), console.warn()***).
- **alert, confirm, prompt:** Métodos para interactuar con el usuario mediante cuadros de diálogo.



NOTA: en algunos sitios se hace referencia a los elementos del **BOM** como **Web API** de JavaScript.

Tipos de nodos en el DOM

En el árbol **DOM**, cada elemento de una página HTML se representa mediante un tipo de nodo específico. Los principales tipos de nodo son:

1. Nodos de Elemento (**ELEMENT_NODE**):

Representan las etiquetas HTML, como `<div>`,`<h1>`,`<p>`, etc.

Su `nodeType` es 1.

2. Nodos de Atributo (**ATTRIBUTE_NODE**):

Representan los atributos de un nodo de elemento, como `class="mensaje"` o `id="titulo"`.

Estos nodos no están directamente en el árbol DOM, pero se pueden acceder a través de sus elementos.

Su `nodeType` es 2.

3. Nodos de Texto (**TEXT_NODE**):

Contienen el texto dentro de los elementos HTML.

En el DOM, cada fragmento de texto tiene su propio nodo de texto, incluso si está dentro de un solo nodo de elemento.

Su nodeType es 3.

4. Nodos de Comentario (**COMMENT_NODE**):

Representan comentarios en el HTML, como `<!-- Comentario -->`.

Pueden ser útiles para almacenar notas que no se mostrarán en la página.

Su nodeType es 8.

5. Nodo Documento (**DOCUMENT_NODE**):

Es la raíz de todo el árbol DOM.

Representa el documento HTML completo.

Su nodeType es 9.

6. Nodo de Fragmento de Documento (**DOCUMENT_FRAGMENT_NODE**).

Es un nodo documento sin padre, no está vinculado a la página.

Es útil para componer parte de la página de manera independiente, una vez concluida la edición podemos integrar el nodo en la estructura de nuestra página.

Su nodeType es 11.

Diferencias entre HTMLCollection y NodeList

Conviene que antes de comenzar a manipular el DOM veamos los distintos tipos de listas de nodos que podemos encontrarnos. Habitualmente nos encontramos con dos tipos de colección "***HTMLCollection***" y "***NodeList***", veamos las características de cada una.

HTMLCollection

Contiene una colección de nodos que cumplen con un criterio especifico. Por ejemplo todos los elementos de tipo "*** ***" obtenidos con "***getElementsByTagName('div')***".

La colección está "**viva**", es decir, si aparecen o desaparecen elementos que cumplen el criterio de filtrado la colección se actualiza automáticamente.

No puede recorrerse con métodos como "***.forEach()***" o similares. Hay que usar la notación array [] con un índice o un nombre de elemento.

NodeList

Contiene una colección de nodos de cualquier tipo, no sólo etiquetas HTML.

Puede ser una colección estática o viva según como se cree o se acceda.

- Si empleamos un método de búsqueda, por ejemplo "***querySelectorAll()***", la lista devuelta es estática, no cambia.
- Si accedemos a través de la propiedad "***childNodes***" de un elemento HTML, la lista está viva.

Puede recorrerse con métodos como "***.forEach()***" o similares.

Navegación entre nodos

Una vez que el código HTML se convierte en un árbol DOM, podemos utilizar diversas propiedades y métodos para navegar entre los nodos.

1. parentNode

Devuelve el nodo padre de un nodo dado.

Si el nodo no tiene un padre (por ejemplo, el "***document***"), devolverá null.

```
const nodo = document.getElementById('titulo');
console.log(nodo.parentNode); // <body> (contenido NODO)
```

2. childNodes

Devuelve una "***NodeList***" con todos los nodos hijos (incluyendo nodos de texto y comentarios).

Es útil para acceder a todos los hijos de un nodo.

```
const cuerpo = document.body;
console.log(cuerpo.childNodes); // Lista de nodos hijos del <body>
(contenido NODO)
```

3. children

Similar a "***childNodes***", pero solo devuelve nodos de tipo elemento, excluyendo nodos de texto y comentarios.

```
const lista = document.getElementById('miLista');
console.log(lista.children); // Lista de elementos hijos (contenido NODO)
```

4. firstChild y lastChild

"***firstChild***" devuelve el primer nodo hijo, y "***lastChild***" devuelve el último nodo hijo.

Ten en cuenta que pueden devolver nodos de texto.

```
const parrafo = document.querySelector('p');
```

```
console.log(parrafo.firstChild); // Primer nodo hijo (puede ser un nodo
de texto)
```

5. firstElementChild y lastElementChild

Similares a "***firstChild***" y "***lastChild***", pero solo devuelven nodos de tipo elemento.

```
const contenedor = document.querySelector('.contenedor');
console.log(contenedor.firstElementChild); // Primer nodo elemento hijo
```

6. nextSibling y previousSibling

"***nextSibling***" devuelve el hermano siguiente del nodo actual, y "***previousSibling***" devuelve el hermano anterior.

Estos métodos devuelven cualquier tipo de nodo (incluyendo nodos de texto y comentarios).

```
const item = document.querySelector('li');
console.log(item.nextSibling); // Nodo siguiente (puede ser un nodo de
texto)
```

7. nextElementSibling y previousElementSibling

Similares a "***nextSibling***" y "***previousSibling***", pero solo devuelven nodos de tipo elemento.

```
const elemento = document.querySelector('li');
console.log(elemento.nextElementSibling); // Hermano siguiente de tipo
elemento
```

Ejemplo navegación entre nodos.

```
<!DOCTYPE html>
<html lang="es">
<head>
    <title>Navegación entre nodos - Ejemplo</title>
</head>
<body>
    <h1 id="titulo">Bienvenido</h1>
    <ul id="miLista">
        <li>Elemento 1</li>
        <li>Elemento 2</li>
        <li>Elemento 3</li>
    </ul>
    <p>Este es un párrafo.</p>
</body>
</html>
```

Y el código para navegar entre los nodos.

```
// Obtener el nodo <h1>
const titulo = document.getElementById('titulo');

// Navegar al nodo padre
console.log(titulo.parentNode); // <body> (contenido NODO)

// Navegar al siguiente nodo
console.log(titulo.nextElementSibling); // <ul id="miLista"> (contenido
NODO)

// Obtener la lista de nodos hijos del <ul>
const lista = document.getElementById('miLista');
console.log(lista.children); // NodeList [<li>, <li>, <li>] (contenido
NODO)

// Obtener el primer y último hijo de la lista
console.log(lista.firstElementChild.textContent); // "Elemento 1"
console.log(lista.lastElementChild.textContent); // "Elemento 3"

// Navegar entre elementos hermanos
const primerElemento = lista.firstElementChild;
console.log(primerElemento.nextElementSibling.textContent);//"Elemento 2"
```

Búsqueda de Nodos

Además de navegar usando las propiedades anteriores, también podemos buscar elementos de forma directa en el árbol DOM utilizando algunos métodos muy útiles:

1. getElementById()

Devuelve el elemento con un id específico.

```
const elemento = document.getElementById('titulo');
```

2. getElementsByClassName()

Devuelve una "***HTMLCollection***" con todos los elementos que tienen la clase especificada.

```
const elementos = document.getElementsByClassName('miClase');
```

3. getElementsByTagName()

Devuelve una "***HTMLCollection***" con todos los elementos que coinciden con el nombre de la etiqueta.

```
const parrafos = document.getElementsByTagName('p');
```

4. querySelector()

Devuelve el primer elemento que coincide con un selector CSS.

```
const parrafo = document.querySelector('p');
```

5. querySelectorAll()

Devuelve una "***NodeList***" con todos los elementos que coinciden con un selector CSS.

```
const items = document.querySelectorAll('li');
```

6. closest()

Busca hacia arriba en el árbol DOM el primer ancestro que coincide con un selector CSS.

```
const elemento = document.querySelector('span');
const contenedor = elemento.closest('.contenedor');
```

Ejemplo de búsqueda de nodos.

```
<!DOCTYPE html>
<html lang="es">
<head>
    <title>Búsqueda DOM</title>
</head>
<body>
    <div class="contenedor">
        <span class="texto">Hola</span>
    </div>
</body>
</html>
```

Y el código para buscar nodos.

```
// Obtener el elemento por su clase
const span = document.querySelector('.texto');

// Buscar el contenedor más cercano hacia arriba en el árbol
const contenedor = span.closest('.contenedor');
console.log(contenedor); // <div class="contenedor">
```

Creación y manipulación de nodos DOM

Una de principales capacidades de JavaScript es la de manipular el árbol DOM en tiempo de ejecución evitando tener que recargar la página. En este apartado vamos a ver cómo crear elementos, modificar sus atributos, añadir clases, y trabajar con el atributo **"*data-**"** para almacenar metadatos.

Crear Nodos

Los nuevos nodos los crea siempre el objeto "`document`". Habitualmente vamos a crear elementos HTML pero recuerda que disponemos de métodos específicos para otros tipos de nodo.

- **document.createElement('etiqueta HTML')**: crea un nuevo nodo de tipo elemento HTML (etiqueta HTML).
- **document.createTextNode('texto plano')**: crea un fragmento de texto sin envolverlo en elemento HTML.
- **document.createComment('texto comentario')**: crea un comentario que se verá en el código de la página.
- **document.createDocumentFragment()**: crea un nodo virtual. Útil para cuando queremos crear una colección de nodos sin nodo padre, ya que el nodo virtual desaparece al insertarse en el DOM.

Por ejemplo.

```
// Crear un nuevo elemento <p>
const nuevoParrafo = document.createElement('p');
nuevoParrafo.textContent = 'Este es un párrafo nuevo.';

// Crear un nodo de texto
const texto = document.createTextNode('Este es un texto sin etiqueta.');
document.body.appendChild(texto);

// Crear un comentario
const comentario = document.createComment('Este es un comentario.');
document.body.appendChild(comentario);
```

Propiedades innerHTML y textContent

Adicionalmente a los métodos para crear nodos podemos interactuar directamente con el contenido que cuelga de un nodo con dos propiedades.

- **textContent**: inserta sólo texto, sin interpretar las etiquetas HTML que se incluyan en su interior.
- **innerHTML**: inserta texto HTML, interpretando las etiquetas HTML que se incluyan en su interior.

NOTA: cuidado con "***innerHTML***" ya que puede permitir la inyección de código inseguro.

Insertar, remplazar y eliminar Nodos

Una vez creado un nodo, podemos insertarlo en la página utilizando métodos como:

1. appendChild(): Añade un nodo como último hijo de un elemento.

```
const contenedor = document.getElementById('contenedor');
contenedor.appendChild(nuevoParrafo);
```

2. insertBefore(): Inserta un nodo antes de un nodo específico.

```
const referencia = document.querySelector('.referencia');
contenedor.insertBefore(nuevoParrafo, referencia);
```

3. prepend() (ES6+): Inserta un nodo como primer hijo de un elemento.

```
contenedor.prepend(nuevoParrafo);
```

4. replaceChild(): Reemplaza un nodo existente con otro.

```
const viejoParrafo = document.getElementById('viejo');
contenedor.replaceChild(nuevoParrafo, viejoParrafo);
```

5. remove(): Elimina el propio nodo.

```
nodoBorrar.remove();
```

6. removeChild(): Elimina un nodo hijo.

```
contenedor.removeChild(viejoParrafo);
```

Por ejemplo, a partir del código HTML

```
<body>
    <div id="contenedor">
        <p class="referencia">JavaScript mola</p>
    </div>
</body>
```

Le aplicamos el código JS.

```
const nuevoParrafo = document.createElement('h1');
nuevoParrafo.textContent = 'Mensaje importante';

// Seleccionar el contenedor
const contenedor = document.getElementById('contenedor');

// Insertar el nuevo título antes del párrafo de referencia
const referencia = document.querySelector('.referencia');
contenedor.insertBefore(nuevoParrafo, referencia);
```

Genera la siguiente imagen.

Manipulación de atributos y clases

Una vez creado un nodo, podemos asignarle atributos y clases utilizando los siguientes métodos:

1. setAttribute('nombre_atr', 'valor'): Asigna un atributo a un elemento.

```
nuevoParrafo.setAttribute('id', 'nuevo');
nuevoParrafo.setAttribute('class', 'mi-clase');
```

2. getAttribute('nombre_atr'): Obtiene el valor de un atributo.

```
console.log(nuevoParrafo.getAttribute('id')); // "nuevo"
```

3. removeAttribute('nombre_atr'): Elimina un atributo.

```
nuevoParrafo.removeAttribute('class');
```

NOTA: un elemento HTML dispone de una propiedad "***.attributes***" que incluye todos sus atributos (incluidos "***data-****" y "***class***").

Atributo "class" y "classList"

Las etiquetas HTML emplean el atributo "`class`" con dos finalidades: aplicar estilos CSS o para añadir etiquetas semánticas.

Como ya sabes, este atributo admite una lista de valores separados por un espacio. Por ejemplo, el siguiente párrafo tiene las etiquetas de "noticia" y de "importante".

```
<p class="noticia importante"></p>
```

En JavaScript este atributo se ve como "**classList**" y dispone de métodos especiales para trabajar con esta colección singular.

- **elemento.classList.add('clase')**: Añade una nueva clase a la lista.
- **elemento.classList.remove('clase')**: Elimina una clase de la lista.
- **elemento.classList.toggle('clase'):** Si no contiene la clase la añade, si la contine la elimina.
- **elemento.classList.contains('clase'):** Verdadero si contiene la clase.
- **elemento.classList.replace('claseActual', 'claseNueva'):** Reemplaza una clase por otra.

NOTA: la propiedad "***className***" contiene una cadena con el valor de "***class***" en texto plano. Por ejemplo, "elemento.className='alumno informática segundo' ".

Por ejemplo.

```
const boton = document.createElement('button');
boton.textContent = 'Haz clic aquí';
boton.classList.add('btn', 'btn-primario');
document.body.appendChild(boton);

// Cambiar la clase al hacer clic
boton.addEventListener('click', () => {
    boton.classList.toggle('activo');
});
```

Genera la siguiente salida en el depurador.

```
<button class="btn btn-primario">Haz clic aquí</button>
```

Y

```
<button class="btn btn-primario activo">Haz clic aquí</button>
```

Uso del atributo data-*

Las etiquetas HTML emplean el atributo "***data-****" para añadir metadatos (información relativa al elemento asociado) no visibles en la página.

El atributo "***data-****" nos da gran flexibilidad ya que nos permite definir nuestros propios atributos sin interferir en comportamiento de otros elementos de la página.

```
<div data-id="456" data-categoria="electronica" data-precio="299">
    Televisor LED
</div>
```

En JavaScript este atributo se ve como "***dataset***" y se usa como una colección de pares de clave valor. En este caso la notación del atributo cambia al desaparecer el data, por ejemplo, en una etiqueta HTML tendríamos "***data-entidad='juanito'***" y en JS "***dataset.entidad='juanito'***".

NOTA: los nombres de los atributos de tipo "***data-****" en JavaScript se convierten a "***camelCase***", desapareciendo los guiones intermedios.

Operaciones básicas.

1. element.dataset.nombreAtributo: Acceder a un Atributo.

2. element.dataset.nombreAtributo = valor: Modificar un atributo.

3. element.removeAttribute('nombreAtributo') o **delete element.dataset.nombreAtributo**: Elimina un atributo.

Por ejemplo.

```
<body>
    <div id="producto" data-id="456" data-categoria="electronica" data-
precio="299">
        Televisor LED
    </div>

    <script>
        const producto = document.getElementById('producto');

        // Recorrer todos los atributos data-* del elemento
        for (let key in producto.dataset) {
            console.log(`${key}: ${producto.dataset[key]}`);
        }
    </script>
</body>
```

Que genera la siguiente salida.

```
id: 456
categoria: electronica
precio: 299
```

Eventos en el DOM

Los eventos nos permiten interactuar con la página web. Un evento es una acción que ocurre en la página, dependiendo del tipo de evento producido el evento puede incluir información adicional.

A continuación, vamos a ver como trabajar con eventos, tipos de eventos, delegación de eventos y comprender el flujo de un evento.

Cómo definir eventos

Existen varias formas de definir eventos en JavaScript, y todas se emplean.

1. Atributo HTML (onclick)

La forma más sencilla y directa de definir un evento es a través del atributo HTML del elemento.

Por ejemplo.

```
<button onclick="alert('¡Hola!')">Haz clic aquí</button>
```

Esta forma mezcla el JavaScript con el HTML, lo que no es ideal para mantener un código limpio y modular.

2. Asignación Directa (element.onclick)

Puedes asignar un evento directamente a una propiedad del elemento desde JavaScript.

Por ejemplo.

```
<button id="boton">Haz clic aquí</button>
<script>
    const boton = document.getElementById('boton');
    boton.onclick = function () {
        alert('¡Botón clicado!');
    };
</script>
```

IMPORTANTE: Si asignas otro "***onclick***", sobrescribes el anterior manejador.

3. Método addEventListener()

La forma más flexible y recomendada para gestionar eventos es mediante el uso de "***addEventListener()***". Esto permite registrar múltiples eventos en un mismo elemento sin sobrescribir los anteriores.

Por ejemplo.

```
<button id="boton">Haz clic aquí</button>
<script>
    const boton = document.getElementById('boton');
    boton.addEventListener('click', () => {
        alert('Evento con addEventListener');
    });
</script>
```

Esta manera de definir eventos permite añadir múltiples manejadores al mismo evento, además facilita la eliminación de manejadores específicos.

Eliminar eventos dinámicamente

Podemos eliminar un evento con **"*removeEventListener()*"**.

IMPORTANTE: Dependiendo del método empleado para definir el evento es posible que no sea posible eliminar el evento.

Por ejemplo.

```
<button id="boton3">Haz clic aquí</button>
<script>
    function mostrarAlerta() {
        alert('Evento activo');
    }

    const boton3 = document.getElementById('boton3');
    boton3.addEventListener('click', mostrarAlerta);

    // Eliminar el evento después de 3 segundos
    setTimeout(() => {
        boton3.removeEventListener('click', mostrarAlerta);
        alert('Evento eliminado');
    }, 3000);
</script>
```

Tipos de Eventos Comunes

A continuación, se presentan algunos de los eventos más comunes, junto con las propiedades útiles del objeto **"evento"** que puedes utilizar en tus manejadores.

NOTA: A continuación sólo se van a desarrollar una parte de la teoría de eventos en JavaScript que el autor ha considerado más relevante. Ten en mente que hay otros eventos útiles que se omiten como "scroll", gestión de "drag and drop", gestión del portapapeles (clipboard), o la reproducción multimedia.

Propiedades comunes del objeto "evento"

El objeto ***"evento"*** que se pasa automáticamente a los manejadores de eventos contiene varias propiedades útiles:

- **evento.target**: El elemento en el que se originó el evento.
- **evento.currentTarget**: El elemento al que se asignó el evento (útil en delegación de eventos donde tenemos un contenedor con subelemento donde capturamos el evento).
- **evento.type**: El tipo de evento (*click*, *keydown*, etc.).
- **evento.timeStamp**: Devuelve el momento en el que se creó el evento.

Eventos del documento (y de la ventana)

DOMContentLoad

Se dispara cuando el HTML ha sido completamente cargado y parseado, pero antes de que las imágenes y otros recursos externos se hayan cargado.

Por ejemplo.

```
document.addEventListener('DOMContentLoaded', () => {
```

```
    console.log('El DOM está listo');
});
```

window.onload

Se dispara sólo cuando todos los recursos de la página (imágenes, scripts, etc.) se han cargado.

Por ejemplo.

```
window.onload = () => {
    console.log('Página completamente cargada');
};
```

Eventos de Ratón

Además de los eventos como "**click**", "**dblclick**", "**contextmenu**" (pulsar botón derecho), "**mouseover**", "**mouseout**", "**mouseup**", "**mousedown**", etc., el objeto "**event**" (realmente **MouseEvent**) para eventos de ratón tiene propiedades útiles:

- **"event.button"**: Indica qué botón del ratón se ha pulsado (0 = izquierdo, 1 = medio, 2 = derecho).
- **"event.clientX"** / **"event.clientY"**: Coordenadas del ratón respecto a la ventana.
- **"event.pageX"** / **"event.pageY"**: Coordenadas del ratón respecto a la página completa.

Por ejemplo.

```
<div id="area" style="width:200px; height:100px; background-
color:lightgray;">Área de clic</div>
<script>
    const area = document.getElementById('area');
    area.addEventListener('click', (event) => {
        console.log(`Botón: ${event.button}, X: ${event.clientX}, Y:
${event.clientY}`);
    });
</script>
```

Eventos de teclado

Los eventos de teclado (***keydown***, ***keyup***) permiten detectar pulsaciones de teclas:

- **event.key**: Devuelve la tecla presionada ("a", "Enter", etc.).
- **event.code**: El código físico de la tecla ("KeyA", "Space", etc.).
- **event.ctrlKey, event.shiftKey, event.altKey, event.metaKey**: Detecta si se están presionando teclas modificadoras.

NOTA: "***metaKey***" es la tecla Windows o Cmd en Mac.

Por ejemplo.

```
document.addEventListener('keydown', (event) => {
    console.log(`Tecla: ${event.key}, Código: ${event.code}`);
    if (event.ctrlKey && event.key === 's') {
        event.preventDefault();
        console.log('Guardando...');
    }
});
```

Eventos de formulario

Los formularios tienen varios eventos útiles para validar y gestionar la entrada de datos:

- **submit**: Se dispara cuando el formulario se envía.
- **reset**: Se dispara cuando se restablece el formulario.
- **input**: Se dispara cada vez que cambia el valor de un campo.
- **change**: Se dispara cuando un campo pierde el foco tras un cambio.
- **input**: Se dispara cuando un campo cambia de valor (se dispara en edición con cada cambio).
- **focus** / **blur**: Cuando un campo recibe o pierde el foco (útiles para validaciones dinámicas).
- **invalid**: Se dispara cuando un campo no cumple con su validación.

Por ejemplo.

```
<form id="miFormulario">
    <input type="text" id="nombre" required>
    <input type="email" id="correo" required>
    <button type="submit">Enviar</button>
    <button type="reset">Restablecer</button>
</form>
<script>
    const formulario = document.getElementById('miFormulario');
    formulario.addEventListener('submit', (event) => {
        event.preventDefault(); // Evita el envío por defecto
        alert('Formulario enviado');
    });
    formulario.addEventListener('reset', () => {
        console.log('Formulario restablecido');
    });
    const correo = document.getElementById('correo');
    correo.addEventListener('invalid', () => {
        alert('Por favor, ingresa un correo válido.');
    });
</script>
```

Delegación de eventos

La **delegación de eventos** es una técnica que consiste en asignar un evento a un contenedor en lugar de asignarlo a cada uno de sus elementos hijos. Esto mejora el rendimiento y simplifica el código, especialmente cuando se trabaja con listas largas o elementos generados dinámicamente.

Por ejemplo.

```
<ul id="lista">
    <li>Elemento 1</li>
    <li>Elemento 2</li>
    <li>Elemento 3</li>
</ul>
<script>
    const lista = document.getElementById('lista');
    lista.addEventListener('click', (event) => {
        if (event.target.tagName === 'LI') {
            alert(`Has clicado en: ${event.target.textContent}`);
        }
    });
</script>
```

Flujo de eventos: Bubbling y Capturing

El flujo de eventos en el DOM ocurre en dos fases conocidas como "***capturing***" y "***bubbling***".

1. Capturing (Fase de captura):

El evento se propaga desde el elemento más externo (***window***) hasta el objetivo (***target***).

2. Bubbling (Fase de burbuja):

Una vez que el evento llega al objetivo, se propaga hacia afuera, de regreso a ***window***.

Por defecto, los eventos se manejan en la fase de burbuja. Sin embargo, puedes gestionar la fase de captura pasando un tercer parámetro **true** en "***addEventListener()***".

Por ejemplo.

```
<div id="contenedor">
    <button id="boton">Pulsame</button>
</div>
<script>
    const contenedor = document.getElementById('contenedor');
    const boton = document.getElementById('boton');

    contenedor.addEventListener('click', () => {
        console.log('Contenedor clicado');
    }, true); // Usar captura, por defecto false

    boton.addEventListener('click', () => {
```

```
        console.log('Botón clicado');
    });
</script>
```

Que genera la siguiente salida.

```
Contenedor clicado
Botón clicado
```

Podemos gestionar el evento una vez disparado mediante varios métodos:

- **event.preventDefault()**: Evita el comportamiento por defecto del evento (por ejemplo, no seguir el enlace).
- **event.stopPropagation()**: Detiene la propagación del evento (***bubbling***).
- **event.stopImmediatePropagation()**: Detiene la propagación y evita que otros manejadores del mismo evento se ejecuten. (Detiene TODOS los eventos cuando hay múltiples manejadores asociados al mismo evento).

Por ejemplo.

```
<a href="https://www.google.com" id="enlace">Ir a Google</a>
<script>
    const enlace = document.getElementById('enlace');
    enlace.addEventListener('click', (event) => {
        event.preventDefault(); // Evita que el enlace se abra
        event.stopPropagation(); // Detiene la propagación del evento
        alert('Enlace bloqueado');
    });
</script>
```

Manejo de formularios

En JavaScript, trabajar con formularios es esencial para acceder y manipular los datos que los usuarios ingresan. El DOM nos ofrece herramientas como ***"document.forms"*** y ***"form.elements"*** para interactuar con los formularios y sus campos de manera estructurada.

Colecciones document.forms y form.elements

"document.forms" es una colección de todos los formularios de la página, accesibles por su índice (document.forms[0]) o por su atributo "name" (document.forms['miFormulario']).

Cada formulario tiene una propiedad "**elements**", que es una colección de todos los campos del formulario, como "***<input>***", "***<textarea>***", "***<select>***", o "***<button>***".

Por ejemplo

```
<form name="miFormulario">
    <input type="text" name="nombre" value="Juan">
    <input type="email" name="correo" placeholder="Ingresa tu correo"
value="juan@example.com">
    <button type="submit">Enviar</button>
</form>

<script>
    const formulario = document.forms['miFormulario'];

    // Acceder a los campos por su índice
    console.log(formulario.elements[0].name); // "nombre"

    // Acceder a los campos por su nombre
    const campoCorreo = formulario.elements['correo'];
    console.log(campoCorreo.placeholder); // "Ingresa tu correo"

    // Recorrer todos los elementos del formulario
    for (let elemento of formulario.elements) {
        console.log(`Elemento: ${elemento.name}, Tipo: ${elemento.type},
Valor: ${elemento.value}`);
    }
</script>
```

Que genera la siguiente salida.

```
nombre
Ingresa tu correo
Elemento: nombre, Tipo: text, Valor: Juan
Elemento: correo, Tipo: email, Valor: juan@example.com
Elemento: , Tipo: submit, Valor:
```

Estas dos propiedades nos permiten trabajar fácilmente con los formularios y elementos del formulario.

Propiedades y métodos de un formulario

Los formularios tienen varias propiedades clave que definen cómo interactúan con el servidor, como su "***action***", "***method***", y "***enctype***". Estas propiedades se pueden leer y modificar dinámicamente con JavaScript para cambiar su comportamiento en tiempo de ejecución.

1. action: Representa la URL a la que se enviarán los datos del formulario.

2. method: Método HTTP utilizado en el envío (GET o POST), valor por defecto GET.

3. enctype: Define como se codifican los datos al enviarlos al servidor. Valores posibles:

- application/x-www-form-urlencoded (por defecto).
- multipart/form-data (para subir archivos).
- text/plain (menos común).

4. target: Especifica donde se abre el formulario. Valores posibles:

- **_self**: El resultado se muestra en la ventana actual (por defecto).
- **_blank**: El resultado se muestra en una nueva ventana.
- **_parent**: El resultado se muestra en el contexto de navegación padre del marco actual, si no hay un contexto nuevo igual que _self.
- **_top**: El resultado se muestra en el contexto de navegación que es ancestro del actual y no tiene padre, si no hay padre igual que _self.

Por ejemplo.

```
    <form name="miFormulario" action="/submit" method="GET"
target="_blank">
        <input type="text" name="usuario" placeholder="Usuario">
        <button type="submit">Enviar</button>
    </form>

    <script>
        const formulario = document.forms['miFormulario'];

        // Leer propiedades iniciales, se pueden sobrescribir
        console.log(`Action: ${formulario.action}`);
        console.log(`Method: ${formulario.method}`);
        console.log(`Enctype: ${formulario.enctype}`);
        console.log(`Target: ${formulario.target}`);
    </script>
```

Que genera las siguiente salida.

```
Action: http://127.0.0.1:5500/submit
Method: get
Enctype: application/x-www-form-urlencoded
Target: _blank
```

Métodos de formulario

Si fuera necesario, mediante JavaScript podemos enviar o restablecer el formulario.

- formulario.**submit()**
- formulario.**reset()**

Validaciones básicas en campos de formulario

HTML ofrece una serie de **atributos** y **pseudoclases** que permiten realizar validaciones básicas en los campos de entrada sin necesidad de escribir código JavaScript. Esto simplifica la validación inicial de formularios y mejora la experiencia del usuario.

Atributos de validación en campos

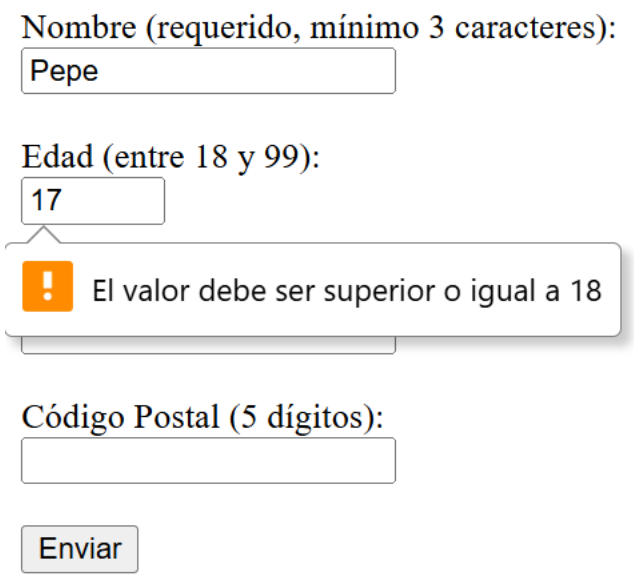
Los siguientes atributos se pueden utilizar para establecer restricciones en los campos de entrada:

1. **required**: (De tipo booleano) Indica que el campo es obligatorio.
2. **min** y **max**: Definen el rango permitido de valores para campos numéricos o fechas.
3. **maxlength** y **minlength**: Establecen el número máximo y mínimo de caracteres permitidos.
4. **pattern**: Define una expresión regular que el campo debe cumplir.
5. **step**: Define el incremento válido para valores numéricos.
6. **placeholder**: Proporciona un texto indicativo dentro del campo (sólo información, no genera validación ni valor por defecto).

Por ejemplo.

```
<form name="miFormulario">
    <label for="nombre">Nombre (requerido, mínimo 3
caracteres):</label><br>
    <input type="text" id="nombre" name="nombre" required
minlength="3"><br><br>

    <label for="edad">Edad (entre 18 y 99):</label><br>
    <input type="number" id="edad" name="edad" min="18" max="99"><br><br>

    <label for="correo">Correo (formato válido):</label><br>
    <input type="email" id="correo" name="correo" required><br><br>

    <label for="codigo">Código Postal (5 dígitos):</label><br>
    <input type="text" id="codigo" name="codigoPostal" pattern="\d{5}"
required><br><br>

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

Que genera la siguiente salida.



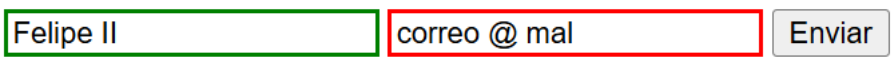
Pseudoclases CSS para la validación.

HTML permite utilizar las pseudoclases "***:valid***" y "***:invalid***" para aplicar estilos dinámicos según si el campo cumple con las restricciones.

Por ejemplo.

```
<form name="miFormulario">
    <input type="text" name="nombre" required minlength="3"
placeholder="Nombre">
    <input type="email" name="correo" required placeholder="Correo">
    <button type="submit">Enviar</button>
</form>

<style>
    input:valid {
        border: 2px solid green;
    }

    input:invalid {
        border: 2px solid red;
    }
</style>
```

Que genera la siguiente salida.



Fíjate que cuando el valor del campo es válido el borde es verde y cuando es invalido el borde es rojo. Si pulsamos enviar se muestra un mensaje informativo para el campo erróneo.

Métodos y validaciones adicionales con JavaScript

Aunque los atributos HTML y las pseudoclases de validación son útiles para realizar validaciones básicas, a menudo es necesario complementar esto con validaciones personalizadas usando JavaScript. Aquí exploraremos cómo utilizar los métodos ***"checkValidity()"*** y ***"setCustomValidity()"*** para añadir validaciones avanzadas y mensajes personalizados.

También contamos con la propiedad ***".validity"*** que contiene información detallada sobre el estado de la validación del elemento de entrada incluyendo validaciones automáticas como personalizadas. OJO, el mensaje de error a mostrar al usuario se encuentra en otra propiedad hermana de sólo lectura ***".validationMessage"***.

Método checkValidity()

El método "***checkValidity()***" permite verificar si un campo o un formulario cumple con todas las restricciones de validación definidas en sus atributos.

Este método está disponible tanto a **nivel de campo como a nivel de formulario**. Cuando se ejecuta a nivel de formulario se comprueba que todos los campos sean válidos.

Por ejemplo.

NOTA: El siguiente ejemplo tiene código comentado que no funciona, el "***submit***" no se dispara si el formulario no es válido.

```
<form id="miFormulario">
    <input type="text" id="nombre" name="nombre" required minlength="3"
placeholder="Nombre">
    <!-- <button type="submit">Enviar</button> -->
    <input type="button" value="Enviar"/>
</form>

<script>
    const formulario = document.getElementById('miFormulario');
    const campoNombre = document.getElementById('nombre');
    const boton = document.querySelector('[type=button]');

    /* El submit no se lanza si algún campo no es válido */
    // formulario.addEventListener('submit', (event) => {
    boton.addEventListener('click', (event) => {
        if (!campoNombre.checkValidity()) {
            //event.preventDefault();
            alert('El nombre no cumple con las restricciones.');
            return;
        }
        formulario.submit();
    });
```

```
</script>
```

Método setCustomValidity()

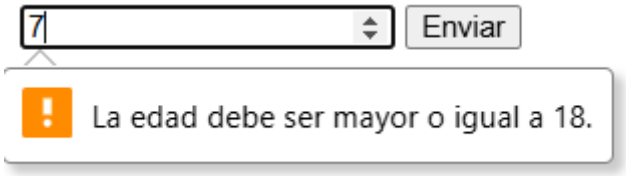
El método ***"setCustomValidity()"*** permite establecer un mensaje de error personalizado en un campo. Este mensaje será mostrado automáticamente por el navegador si el campo no es válido.

IMPORTANTE: si la cadena de error personalizado está vacía el campo es válido, cualquier otro valor en la cadena el campo es invalido.

Por ejemplo.

```
<form id="miFormulario">
    <!-- <input type="number" id="edad" name="edad" min="18" max="99" placeholder="Edad (entre 18 y 99)" required> -->
    <input type="number" id="edad" name="edad" placeholder="Edad (entre 18 y 99)" required>
    <button type="submit">Enviar</button>
</form>

<script>
    const campoEdad = document.getElementById('edad');

    campoEdad.addEventListener('input', () => {
        // Reseteamos mensajes de error previos
        campoEdad.setCustomValidity('');
        let valor = parseInt(campoEdad.value);
        if (isNaN(valor)) {
            return;
        }
        if (valor < 18) {
            campoEdad.setCustomValidity('La edad debe ser mayor o igual a 18.');
        } else if (valor > 99) {
            campoEdad.setCustomValidity('La edad debe ser menor o igual a 99.');
        }

        /* Objeto con el estado de la validación */
        console.log(campoEdad.validity);
    });
</script>
```

Que genera la siguiente salida.



Ejemplo de validación con resumen externo

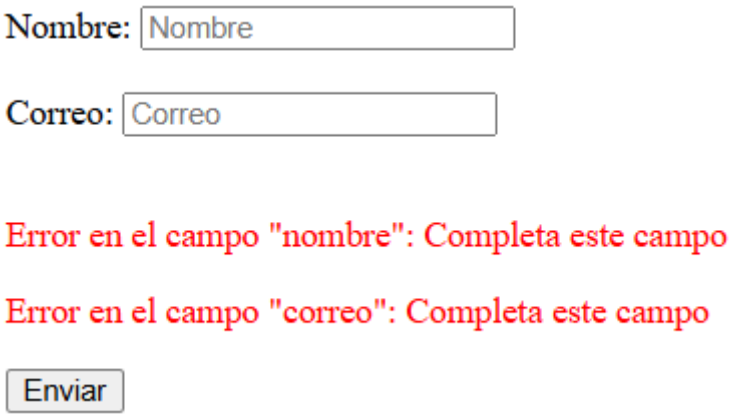
En este ejemplo vamos comprobar los errores de validación cada vez que abandonemos (evento "***blur***") un campo y vamos a mostrar el resumen de errores agrupado en un recuadro apartado.

Por ejemplo

```
<form id="miFormulario">
    <label for="nombre">Nombre:</label>
    <input type="text" id="nombre" name="nombre" required minlength="3"
placeholder="Nombre"><br><br>
    <label for="correo">Correo:</label>
    <input type="email" id="correo" name="correo" required
placeholder="Correo"><br><br>

    <div id="errores" style="color: red;"></div>

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

<script>
    const formulario = document.getElementById('miFormulario');
    const errores = document.getElementById('errores');

    Array.from(formulario.elements).forEach(element => {
        element.addEventListener('blur', () => {
            let esValido = true;
            let mensaje = '';
            for (let campo of formulario.elements) {
                if (!campo.checkValidity()) {
                    esValido = false;
                    mensaje += `<p>Error en el campo "${campo.name}":
${campo.validationMessage}</p>`;
                }
            };

            if(esValido){
                errores.innerHTML = '';
            }else{
                errores.innerHTML = mensaje;
            }

        })
```

```
    });
</script>
```

Que genera la siguiente salida.



NOTA: se podría optimizar para sólo listar los errores de los campos en los que previamente se ha puesto el foco.

Ejemplo de validación con resumen en campo

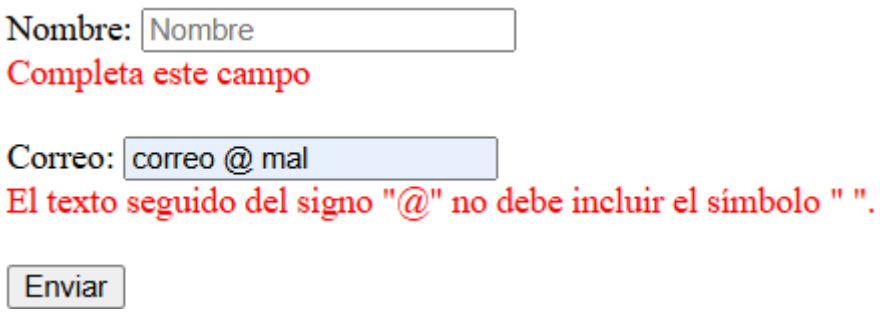
La validación con resumen a nivel campo permite tener una retroalimentación inmediata y clara al usuario. En este caso cada campo a validar cuenta con un área de error propio próximo a la entrada.

Por ejemplo.

```
<form id="miFormulario">
    <label for="nombre">Nombre:</label>
    <input type="text" id="nombre" name="nombre" required minlength="3"
placeholder="Nombre"><br>
    <span id="errorNombre" style="color: red;"></span><br><br>

    <label for="correo">Correo:</label>
    <input type="email" id="correo" name="correo" required
placeholder="Correo"><br>
    <span id="errorCorreo" style="color: red;"></span><br><br>

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

<script>
    const campoNombre = document.getElementById('nombre');
    const campoCorreo = document.getElementById('correo');
    const errorNombre = document.getElementById('errorNombre');
    const errorCorreo = document.getElementById('errorCorreo');
```

```
    campoNombre.addEventListener('input', () => {
        if (!campoNombre.checkValidity()) {
            errorNombre.textContent = campoNombre.validationMessage;
        } else {
            errorNombre.textContent = '';
        }
    });

    campoCorreo.addEventListener('input', () => {
        if (!campoCorreo.checkValidity()) {
            errorCorreo.textContent = campoCorreo.validationMessage;
        } else {
            errorCorreo.textContent = '';
        }
    });
</script>
```

Que genera la siguiente salida.



Formulario con múltiples detalles

Este apartado aborda cómo trabajar con formularios dinámicos que incluyen dos partes principales: cabecera y detalles. La cabecera contiene información general (por ejemplo, cliente o fecha), mientras que los detalles son campos que el usuario puede añadir o eliminar dinámicamente (por ejemplo, líneas de pedido o ítems).

Este ejemplo se compone de dos partes, el código HTML con el formulario donde disponemos de un contenedor para los detalles dinámicos, y el script que genera dinámicamente los detalles.

NOTA: la web "https://jsonplaceholder.typicode.com/" proporciona una API gratuita para prototipado y pruebas, cualquier petición correcta que la hagamos devuelve una respuesta exitosa.

Por ejemplo.

```
<form id="miFormulario"
action="https://jsonplaceholder.typicode.com/posts" method="POST">
    <!-- Cabecera -->
```

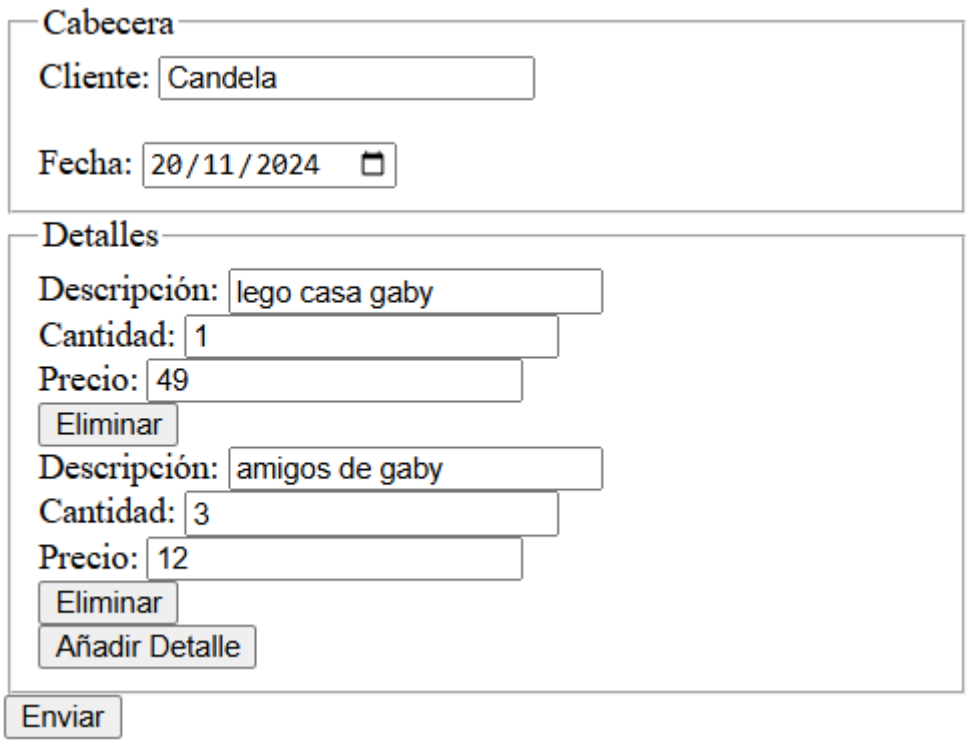
```
    <fieldset>
        <legend>Cabecera</legend>
        <label for="cliente">Cliente:</label>
        <input type="text" id="cliente" name="cliente" required
placeholder="Nombre del cliente"><br><br>

        <label for="fecha">Fecha:</label>
        <input type="date" id="fecha" name="fecha" required><br>
    </fieldset>

    <!-- Detalles -->
    <fieldset>
        <legend>Detalles</legend>
        <div id="contenedorDetalles">
            <!-- Filas dinámicas aquí -->
        </div>
        <button type="button" id="agregarDetalle">Añadir Detalle</button>
    </fieldset>

    <!-- Enviar -->
    <button type="submit">Enviar</button>
</form>
```

Y la parte de código.

```
<script>
    const contenedorDetalles =
document.getElementById('contenedorDetalles');
    const botonAgregar = document.getElementById('agregarDetalle');

    // Función para añadir una fila de detalle
    function agregarFila() {
        // Crear una fila dinámica usando una cadena HTML
        const nuevaFila = `
            <div class="filaDetalle">
                <div>
                    <label>Descripción:</label>
                    <input type="text" name="descripcion[]" required>
                </div>
                <div>
                    <label>Cantidad:</label>
                    <input type="number" name="cantidad[]" min="1"
required>
                </div>
                <div>
                    <label>Precio:</label>
                    <input type="number" name="precio[]" min="0"
step="0.01" required>
```

```
                </div>
                <div>
                    <button type="button"
class="eliminarDetalle">Eliminar</button>
                </div>
            </div>`;

        // Insertar la nueva fila en el contenedor
        contenedorDetalles.insertAdjacentHTML('beforeend',
nuevaFila);

        // Añadir evento al botón de eliminar de esta fila
        const botonesEliminar =
detallesContainer.querySelectorAll('.eliminarDetalle');
        botonesEliminar[botonesEliminar.length -
1].addEventListener('click', (event) => {
            event.target.closest('.filaDetalle').remove();
        });
    }

    // Evento para añadir filas dinámicamente
    botonAgregar.addEventListener('click', agregarFila);
</script>
```

Que genera la siguiente salida



Si depuramos la respuesta esto es lo que nos devuelve el servidor. Fíjate en cómo ha troceado los detalles en colecciones independientes. Hay otro formato alternativo para

trabajar con listas de objetos algo más complejo, pero su uso dependerá del parseador que usemos en el lado del servidor.

```
{
  "cliente": "Candela",
  "fecha": "2024-11-20",
  "descripcion[]": [
    "lego casa gaby",
    "amigos de gaby"
  ],
  "cantidad[]": [
    "1",
    "3"
  ],
  "precio[]": [
    "49",
    "12"
  ],
  "id": 101
}
```

NOTA: El formato alternativo organiza las entradas como un array de objetos, indicamos la propiedad contenedora (detalle) y le damos un índice a cada objeto, seguido indicamos el nombre de cada campo con notación de array. El empleo de un formato u otro dependerá del parseador que emplee el servidor.

```
<input type="text" name="detalle[${contador}][descripcion]" required>
<input type="number" name="detalle[${contador}][cantidad]" min="1"
required>
<input type="number" name="detalle[${contador}][precio]" min="0"
step="0.01" required>
```

Trabajar con FormData

El objeto "***FormData***" permite recopilar los datos de un formulario fácilmente y prepararlos para ser enviados mediante ***"fetch()"***. También puedes recorrer las claves y valores del formulario, lo que resulta útil para depuración o manipulación de datos antes del envío.

Por ejemplo, cancelamos el envío por defecto, logeamos los valores (se podrían validar y cancelar la operación) y enviar la petición al servidor.

RECUERDA: los datos que se envían se construyen a partir de las propiedades "***name***", si sólo usas "***id***" y empleas "***FormData***" los campos sin "**name**" no se incluirán.

```
<form id="miFormulario" method="POST">
    <input type="text" name="nombre" placeholder="Nombre" value="Ana"
required>
    <input type="email" name="correo" placeholder="Correo"
value="ana@correo.com" required>
```

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

<script>
    const formulario = document.getElementById('miFormulario');

    formulario.addEventListener('submit', async (event) => {
        event.preventDefault(); // Evitar el envío por defecto

        const formData = new FormData(formulario);

        // Recorrer claves y valores, CODIGO DESMOSTRATIVO
        for (let [clave, valor] of formData.entries()) {
            console.log(`${clave}: ${valor}`);
        }

        // Enviar formulario
        try {
            const respuesta = await
fetch('https://jsonplaceholder.typicode.com/posts', {
                method: 'POST',
                body: formData,
            });

            if (respuesta.ok) {
                const datos = await respuesta.json();
                console.log('Respuesta del servidor:', datos);
            } else {
                console.error('Error al enviar el formulario');
            }
        } catch (error) {
            console.error('Error en la petición:', error);
        }
    });
</script>
```

NOTA: "***formData***" es la manera habitual de componer el cuerpo de la petición a partir de un formulario, pero también es posible asignarle al cuerpo de la petición una cadena JSON directamente. Por ejemplo. "body: JSON.stringfy({nombre:'Pepe', edad:28})".

Trabajando con peticiones GET

Podemos procesar y manipular peticiones de tipo GET con "***URLSearchParams***". Tenemos dos casos de uso: el primero, dada una URL leer los parámetros, y el segundo, a partir de pares de clave valor componer la URL.

Leer parámetros desde una URL.

Supongamos la ruta de partida “http://localhost:80?nombre=Juan&edad=25”.

Tenemos el siguiente código.

```
<script>
    // Obtener los parámetros de la URL
    const parametros = new URLSearchParams(window.location.search);

    // Leer valores individuales
    const nombre = parametros.get('nombre'); // "Juan"
    const edad = parametros.get('edad'); // "25"

    // Recorrer todos los parámetros
    for (let [clave, valor] of parametros.entries()) {
        console.log(`${clave}: ${valor}`);
    }

 /script>
```

Que genera la siguiente salida.

```
nombre: Juan
edad: 25
```

Crear parámetros para una URL.

En el caso de querer componer una URL conociendo los parámetros también podemos la misma clase.

```
<script>
    const parametros2 = new URLSearchParams();

    // Añadir parámetros
    parametros2.append('nombre', 'Ana');
    parametros2.append('edad', '30');

    // Crear la URL final
    const nuevaUrl = `http://localhost?${parametros2.toString()}`;
    console.log(nuevaUrl); // "http://localhost?nombre=Ana&edad=30"

    // Redirigir al usuario a la nueva URL
    // window.location.href = nuevaUrl;

</script>
```

Que genera la siguiente salida.

http://localhost?nombre=Ana&edad=30

Bibliografía

Documentación MDN árbol DOM

https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model

Desarrollo Web en entono cliente - DOM.

https://xxjcaxx.github.io/libro_dwec/dom.html#

JavaScriptInfo - Documento

https://es.javascript.info/document

Documentación MDN lista de eventos estándar.

https://developer.mozilla.org/es/docs/Web/Events

Documentación MDN eventos de ratón.

https://developer.mozilla.org/es/docs/Web/API/MouseEvent

Documentación MDN evento de teclado.

https://developer.mozilla.org/es/docs/Web/API/KeyboardEvent

Cómo Crear un Array u Objeto en Formularios de HTML.

https://elpan.dev/como-crear-un-array-o-objeto-en-formularios-de-html

JSON Placeholder – API falsa y gratuita para desarrollo y prototipado.

https://jsonplaceholder.typicode.com/