

JAVASCRIPT INTERMEDIO

Introducción al lenguaje de
programación



```
...men  
... data, fn, o  
... of types/handlers  
... "object" ) {  
... es-Object, selector, data )  
... typeof selector !== "string" ) {  
// ( types-Object, data )  
data = data || selector;  
selector = undefined;  
} for ( type in types ) {  
    on( elem, type, selector  
} return elem;  
if ( data == null
```



JavaScript Intermedio	3
Introducción al DOM	3
¿Qué es el DOM?	3
Tipos de objetos en el DOM	3
DOM tree.....	4
Acceso al DOM	5
Modificación, eliminación y creación de contenido y atributos	6
Shadow DOM y Virtual DOM.....	8
Eventos.....	10
Método .addEventListener()	11
Manejo de eventos JavaScript	12
Delegación de eventos	13
Propagación de eventos: Bubbling, Capturing y stopPropagation()	13
Animaciones en JavaScript	15
Animaciones con requestAnimationFrame	15
Transiciones y transformaciones CSS controladas por JavaScript.....	16
Drag and Drop	17
Scope en JavaScript	18
Tipos de Scope	18
Desuso de var y preferencia por let y const	19
Control de tiempo en JavaScript.....	19
setTimeout().....	20
setInterval()	20
clearTimeout()	20
clearInterval()	20
Formularios.....	21
Interactuar con formularios	21
Validaciones de formularios en el frontend	22
Peticiones HTTP	23
Métodos HTTP	24
¿Qué es una API?	24
Peticiones y promesas	25

Conectando con el servidor: fetch() y Web APIs esenciales	26
Sintaxis básica de fetch()	26
Manejo de errores con fetch().....	27
Configuración de headers.....	28
Códigos de estado importantes para tener en cuenta	28
Web APIs esenciales	29
localStorage y sessionStorage	29
WebSockets	29
Service Workers (caché y notificaciones push).....	30
Patrones de diseño	32
Module Pattern (Patrón Módulo).....	32
Factory Pattern (Patrón Fábrica).....	33
Observer Pattern (Patrón Observador)	34

JavaScript Intermedio

Introducción al DOM

¿Qué es el DOM?

El Document Object Model o DOM (“Modelo de Objetos del Documento”) es una estructura en árbol que representa los elementos anidados (etiquetas) que conforman un documento HTML. El DOM nace para proporcionar a los programadores la capacidad de poder detectar eventos generados por el usuario y modificar el documento HTML.

Al acceder a dicha estructura, por medio de distintas propiedades y métodos de distintos objetos, seremos capaces de crear, modificar o eliminar etiquetas en nuestra web, cambiar atributos o contenido de los distintos elementos HTML.

Tipos de objetos en el DOM

Cuando trabajas con el DOM, interactúas con varios tipos de objetos que representan la estructura y el contenido de un documento HTML o XML. Los tipos de objetos más comunes en el DOM incluyen:

- **Document:** representa el objeto raíz que contiene el contenido del documento HTML. Es el punto de entrada para la mayoría de las interacciones con el DOM.
- **Element:** representa un nodo en el DOM que corresponde a un elemento HTML. Un *Element* puede ser cualquier etiqueta en el HTML como `<div>`, ``, `<p>`, etc.
- **Text:** representa el contenido de texto dentro de un elemento. Es un nodo hijo de un *Element* y no tiene etiquetas de inicio o fin, solo el texto plano.
- **Attr:** representa un atributo de un elemento HTML. Los atributos de los elementos como *class*, *id*, *src*, se representan mediante objetos de tipo *Attr*.
- **Node:** es la clase base para todos los objetos del DOM. Un *Node* puede ser un *Element*, *Text*, entre otros. Los nodos pueden ser de diferentes tipos y contienen métodos y propiedades para manipularlos.
- **DocumentFragment:** representa un contenedor ligero de nodos que no está asociado directamente con el DOM de la página. Se usa comúnmente para agrupar nodos antes de insertarlos en el DOM.

- **HTMLCollection:** es una colección de elementos del DOM, generalmente devuelta por métodos como *getElementsByTagName*, *getElementsByClassName*, etc. Es una colección en vivo, lo que significa que se actualiza automáticamente si el contenido del DOM cambia.
- **NodeList:** Es una colección similar a *HTMLCollection* que se devuelve de métodos como *querySelectorAll*. A diferencia de *HTMLCollection*, no es necesariamente una colección en vivo.
- **Window:** aunque no es parte directa del DOM, el objeto *Window* representa el entorno global en el que se ejecuta el documento. Contiene propiedades como *document*, *alert()*, etc.

DOM tree

El DOM tree (o “árbol del DOM”) es una representación jerárquica de la estructura de un documento HTML o XML. En otras palabras, es la forma en que el DOM organiza y representa todos los elementos y nodos del documento en una estructura tipo árbol, donde cada nodo corresponde a un componente del documento (etiquetas HTML, texto, atributos, etc.).

Estructura jerárquica:

- El DOM se organiza en una estructura de árbol, donde cada nodo tiene una relación “padre” e “hijo” con otros nodos.
- El nodo raíz es el objeto *document*, que representa todo el documento HTML. A partir de este nodo raíz, se desprenden los diferentes nodos hijos que representan las etiquetas HTML principales, y a su vez, estos nodos pueden tener sus propios hijos, formando así una estructura jerárquica.

Relaciones entre nodos:

- **Nodo padre (parentNode):** es el nodo que contiene a otros nodos dentro de él.
- **Nodo hijo (childNodes):** son los nodos que están directamente contenidos dentro de un nodo padre.
- **Nodo hermano (sibling):** son nodos que comparten el mismo padre.

Si tenemos el siguiente código HTML:

```

<!DOCTYPE html>
<html>
<head>
  <title>Ejemplo DOM</title>
</head>
<body>
  <h1>Hola, DOM</h1>
  <p>Este es un párrafo.</p>
</body>
</html>

```

El árbol del DOM resultante se vería así:

```

document
├── html
│   ├── head
│   │   └── title
│   │       └── "Ejemplo DOM"
│   └── body
│       ├── h1
│       │   └── "Hola, DOM"
│       └── p
│           └── "Este es un párrafo."

```

Esta estructura nos permite acceder, modificar y manipular los elementos del documento mediante JavaScript.

Acceso al DOM

En esta representación estructurada del documento HTML que es el DOM, tenemos distintas formas de acceder a los elementos.

- **getElementById:** este método permite seleccionar un elemento del DOM mediante su id. Dado que el id es único para cada elemento, *getElementById* siempre retornará un solo elemento

```

const titulo = document.getElementById("mi-titulo");
console.log(titulo);

```

- **getElementsByClassName:** devuelve una colección con todos los elementos que tienen la clase (css).

```

const titulo = document.getElementsByClassName("titulo");
console.log(titulo);

```

- **getElementsByName:** devuelve una colección de elementos con un atributo *name* que coincida.
- **querySelector:** con esto podremos seleccionar el primer elemento que coincida con un selector de CSS. Es más flexible que *getElementById* ya que permite usar cualquier selector, como clases, etiquetas o combinaciones de estos.

```
const parrafo = document.querySelector(".mi-clase");
const div = document.querySelector("div");
```

- **querySelectorAll:** igual que *querySelector* pero selecciona todos los elementos que coinciden con un selector CSS.

Modificación, eliminación y creación de contenido y atributos

Una vez que hemos accedido a un elemento del DOM, podemos modificar tanto su contenido como sus atributos.

- **innerHTML:** este método permite cambiar u obtener el contenido HTML de un elemento. Esto significa que puede incluir tanto texto como etiquetas HTML dentro del elemento.

```
const contenido = document.getElementById("miElemento").innerHTML;
document.getElementById("miElemento").innerHTML = "<p>Este es un párrafo dentro de un div</p>";
```

- **textContent:** es similar a *innerHTML*, pero solo afecta al contenido de texto, sin procesar etiquetas HTML. Es útil cuando se desea modificar u obtener solo el texto sin formato.

```
const mensaje = document.getElementById("mensaje");
mensaje.textContent = "Este es el nuevo contenido de texto";
```

- **setAttribute:** este método permite modificar los atributos de un elemento. Se utiliza para agregar o cambiar atributos como *src*, *href*, *class*, entre otros.

```
const imagen = document.getElementById("imagen");
imagen.setAttribute("src", "nueva-imagen.jpg");
```

- **removeAttribute:** elimina el valor de un atributo de un elemento.

```
const boton = document.getElementById("miBoton");
boton.removeAttribute("class");
```

- **getAttribute:** obtiene el valor de un atributo de un elemento.
- **createElement:** este método crea un nuevo elemento HTML. Una vez creado, le damos contenido con *textContent* y podemos añadirlo al DOM usando otros métodos, como *appendChild*.

```
const nuevoParrafo = document.createElement("p");
nuevoParrafo.textContent = "Este es un nuevo párrafo creado dinámicamente";
document.body.appendChild(nuevoParrafo);
```

- **removeChild:** elimina un elemento hijo de otro elemento. Primero, se debe seleccionar el elemento que contiene al hijo y luego llamamos a *removeChild* para eliminar al nodo hijo.

```
const lista = document.getElementById("lista");
const primerElemento = lista.querySelector("li");
lista.removeChild(primerElemento);
```

- **appendChild:** añade un nuevo nodo como hijo de un nodo existente.

```
const nuevoElemento = document.createElement("div");
document.body.appendChild(nuevoElemento);
```

- **replaceChild:** reemplaza un nodo hijo por otro nodo.

```
const nuevoElemento = document.createElement("div");
const viejoElemento = document.getElementById("viejoElemento");
viejoElemento.parentNode.replaceChild(nuevoElemento, viejoElemento);
```

- **addEventListener:** añade un evento a un elemento.

```
const button = document.getElementById("miBoton");
button.addEventListener("click", function(){
    alert("boton clickeado");
});
```

- **removeEventListener:** elimina un evento de un elemento.
- **classList:** permite manipular las clases de un elemento.

```
const elemento = document.getElementById("miElemento");
elemento.classList.add("nuevaClase");
elemento.classList.remove("viejaClase");
elemento.classList.toggle("otraClase");
```

- **style:** permite modificar el estilo CSS de un elemento.


```
const elemento = document.getElementById("miElemento");
elemento.style.color = "red";
elemento.style.fontSize = "20px";
```

Shadow DOM y Virtual DOM

Ambos conceptos están relacionados con la manipulación del DOM, pero que se utilizan para diferentes propósitos en el desarrollo web.

El **Shadow DOM** es una característica que permite encapsular y aislar una parte de un documento HTML, de modo que su estructura, estilo y comportamiento no interfieran ni se vean afectados por el resto del documento. Esto es útil cuando estamos creando componentes web reutilizables, como un *<custom-element>*, y queremos que tengan su propio conjunto de estilos y lógica sin que interactúen directamente con el resto de la página, también se puede usar de manera independiente.

- **Encapsulamiento:** cuando creas un Shadow DOM en un elemento, se crea un subdocumento dentro del elemento, y este subdocumento está aislado del documento principal. El contenido del Shadow DOM puede tener su propio conjunto de estilos, scripts y estructura HTML sin interferir con el DOM global de la página.
- **Acceso al Shadow DOM:** el acceso a los elementos dentro del Shadow DOM está restringido desde fuera de ese ámbito, lo que significa que los estilos y scripts que están en el documento principal no afectan al contenido dentro del Shadow DOM y viceversa.

```
// Crear un Shadow Root dentro de un elemento
let shadowHost = document.querySelector('#miElemento');
let shadowRoot = shadowHost.attachShadow({mode: 'open'}); // o 'closed'
```

En este ejemplo, el modo *open*, indica que el contenido del Shadow DOM sea accesible desde fuera del mismo. Puedes interactuar con el usando JavaScript. Si el modo lo cambiamos a *closed*, el contenido del Shadow DOM no sea accesible desde fuera. El acceso se limita a la propia implementación interna.

```

<div id="miElemento">Este es el elemento anfitrión</div>

<script>
  let shadowHost = document.getElementById('miElemento');
  let shadowRoot = shadowHost.attachShadow({mode: 'open'});

  shadowRoot.innerHTML = `
    <style>
      p {
        color: red;
      }
    </style>
    <p>Esto es un contenido dentro del Shadow DOM</p>
  `;
</script>

```

El *div* con el *id*="miElemento" se convierte en el anfitrión de un Shadow DOM.

Dentro de ese Shadow DOM, se define un contenido (un párrafo *<p>*) y un estilo (*color: red*), que solo afecta a ese contenido, no al contenido del documento principal.

El **Virtual DOM** es un concepto que se utiliza principalmente en bibliotecas y frameworks como React para optimizar el rendimiento de las aplicaciones web. El objetivo del Virtual DOM es hacer que las actualizaciones del DOM sean más rápidas y eficientes al reducir la cantidad de manipulaciones directas del DOM real, que son costosas en términos de rendimiento.

- **Representación en memoria:** El Virtual DOM es una representación en memoria del DOM real. Es una estructura de datos que refleja la jerarquía de los elementos de la interfaz de usuario, pero no está vinculada directamente al DOM de la página.
- **Proceso de actualización:**
 - **Renderizado inicial:** cuando la aplicación se carga por primera vez, el Virtual DOM se crea como una copia del DOM real.
 - **Cambios en el estado de la aplicación:** cuando hay cambios en los datos de la aplicación (por ejemplo, cuando el usuario interactúa con la interfaz), en lugar de actualizar el DOM real inmediatamente, se actualiza el Virtual DOM.
 - **Comparación (Reconciliación):** luego, el Virtual DOM se compara con el DOM real (este proceso se llama **diffing** o **reconciliación**). Solo se calculan las diferencias entre el Virtual DOM y el DOM real.

- **Actualización eficiente:** después de comparar, solo se aplican las actualizaciones necesarias en el DOM real, lo que minimiza el número de manipulaciones directas del DOM, lo que mejora el rendimiento.

Eventos

En JavaScript, los eventos son acciones o sucesos que ocurren en la página web y a los que un script puede responder. Existen diferentes tipos de eventos clasificados según su categoría y el elemento que los desencadena.

1. **Eventos de mouse:** estos eventos se activan cuando el usuario interactúa con el ratón.
 - click: se activa cuando el usuario hace clic en un elemento.
 - dblclick: se dispara cuando el usuario hace doble clic.
 - mousedown: ocurre cuando el botón del mouse se presiona sobre un elemento.
 - mouseup: se dispara cuando se suelta el botón del mouse después de un mousedown.
 - mousemove: se activa cuando el mouse se mueve sobre un elemento.
 - mouseover: se activa cuando el puntero entra en un elemento.
2. **Eventos de teclado:** estos eventos se activan cuando el usuario interactúa con el teclado.
 - keydown: se activa cuando una tecla es presionada.
 - keyup: se dispara cuando se suelta una tecla.
3. **Eventos de formulario:** estos eventos están relacionados con los formularios HTML.
 - submit: se dispara cuando se envía un formulario.
 - change: se activa cuando un input cambia su valor y pierde el foco.
 - input: se dispara cuando el usuario escribe dentro de un input.
 - focus: ocurre cuando un input gana el foco.
 - blur: se activa cuando input pierde el foco.
 - reset: se dispara cuando un formulario es reseteado.
4. **Eventos de carga y recursos:**
 - load: se activa cuando la página o un recurso ha terminado de cargarse.
 - DOMContentLoaded: se dispara cuando el HTML se ha cargado, pero antes de que se carguen las imágenes y CSS.
5. **Eventos de Drag & Drop:** estos eventos permiten arrastrar elementos dentro de la página.
 - dragstart: se activa cuando el usuario empieza a arrastrar un elemento.
 - drag: se dispara mientras el usuario arrastra el elemento.

- **dragenter**: ocurre cuando el elemento arrastrado entra en una zona de destino.
- **dragover**: se dispara cuando el elemento es arrastrado sobre la zona de destino.
- **dragleave**: se activa cuando el elemento sale de la zona de destino.
- **drop**: se dispara cuando el usuario suelta el elemento arrastrado en la zona de destino.
- **dragend**: ocurre cuando el usuario deja de arrastrar un elemento.

Método `.addEventListener()`

El método `.addEventListener()` nos permite empezar a escuchar un evento concreto, y en el caso de que ocurra, ejecutar la función asociada. De forma opcional, se le puede pasar un tercer parámetro *object* con ciertas opciones, que veremos más adelante.

```
elementoAfectado.addEventListener(tipo-de-evento, función-a-ejecutar);
```

Para ver esto en acción, vamos a verlo con un ejemplo más claro:

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

function saludo() {
  alert('hola!');
}

button.addEventListener('click', saludo);
```

- En el primer parámetro indicamos el **nombre del evento**, en nuestro ejemplo, *click*.
- En el segundo parámetro indicamos la **función a ejecutar** al ocurrir el evento.

También, en lugar de tener la función definida fuera, la podemos incluir dentro del propio `.addEventListener()` utilizando una función anónima. Si prefieres utilizar las **funciones flecha** de JavaScript, quedaría incluso más legible:

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

button.addEventListener("click", () => {
  alert("hola!");
});
```

De hecho, cuando tenemos un código muy corto, podemos incluso ahorrarnos las llaves en la arrow function y escribirlo todo en una sola línea.

Manejo de eventos JavaScript

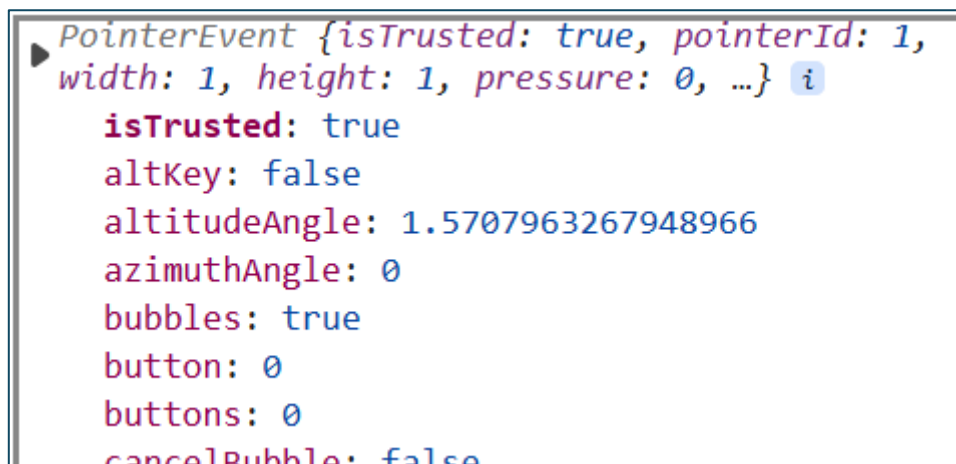
Cuando se disparan ciertos eventos, hay casos en los que nos podría interesar obtener **información relacionada** con la naturaleza del evento en cuestión. Por ejemplo, si estamos escuchando un evento de tipo *click* nos podría interesar saber con qué botón del ratón se ha hecho clic, en qué punto concreto de la pantalla se ha hecho clic, si se estaba pulsando un botón al mismo tiempo mientras se hacía clic, etc.

Hasta ahora, al trabajar con `.addEventListener()`, hemos indicado como segundo parámetro de la función, una arrow function sin parámetros. Sin embargo, podemos definir un parámetro (con el nombre que queramos, por ejemplo, **event**) que nos va a proporcionar información interesante sobre el evento.

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

button.addEventListener("click", (event) => {
  console.log(event);
});
```

El **objeto del evento** es una instancia especial que JavaScript crea automáticamente cuando ocurre un evento. Este objeto va a contener información sobre el evento.



```
PointerEvent {isTrusted: true, pointerId: 1,
width: 1, height: 1, pressure: 0, ...} i
  isTrusted: true
  altKey: false
  altitudeAngle: 1.5707963267948966
  azimuthAngle: 0
  bubbles: true
  button: 0
  buttons: 0
  cancelBubble: false
```

El objeto del evento tiene muchas propiedades útiles dependiendo del tipo de evento que se esté manejando. Aquí están algunas de las más importantes:

- **type**: muestra el tipo de evento que ocurrió.

```
button.addEventListener('click', (e) => {
  console.log(e.type); // "click"
});
```

- **target**: muestra el elemento HTML que disparó el evento.

```
button.addEventListener('click', (e) => {
  console.log(e.target);
});
```

- `currentTarget`: muestra el elemento al que está adjunto el event listener.
- `preventDefault`: Evita el comportamiento por defecto del evento.
- `key`: muestra qué tecla fue presionada (para eventos de teclado).

Delegación de eventos

La delegación de eventos **es una técnica que permite manejar eventos de múltiples elementos hijos con un solo event listener en un elemento padre**. En lugar de agregar un controlador de eventos a cada elemento individual, se coloca en un ancestro común, aprovechando el concepto de propagación de eventos.

¿Cómo funciona? Cuando un evento ocurre en un elemento hijo, este se propaga hacia arriba a través del DOM (bubbling), lo que permite que el evento sea detectado en un ancestro común. Usando esto, podemos comprobar en el controlador si el evento ocurrió en un elemento específico y ejecutar la acción correspondiente.

Supongamos que tenemos una lista `` con varios `` dentro, y queremos manejar los clics en cada `` sin agregar un evento a cada uno.

```
<ul id="lista">
  <li>Elemento 1</li>
  <li>Elemento 2</li>
  <li>Elemento 3</li>
</ul>

<script>
document.getElementById("lista").addEventListener("click", function(event) {
  if (event.target.tagName === "LI") { // Verifica si se hizo clic en un <li>
    alert("Has hecho clic en: " + event.target.textContent);
  }
});
</script>
```

Esto nos permitirá reducir el número de listeners, consumir menos memoria y si se agregasen nuevos elementos al DOM el evento seguirá funcionando sin necesidad de volver a agregar manejadores.

Propagación de eventos: Bubbling, Capturing y stopPropagation()

Cuando ocurre un evento en el DOM, este se propaga en dos fases principales:

1. **Capturing (captura, también llamada “trickling”)**: el evento se propaga desde el documento raíz hacia el elemento objetivo.
2. **Bubbling (burbujeo)**: el evento se propaga en sentido inverso, desde el elemento objetivo hacia arriba por el DOM.

En el **bubbling**, el evento comienza en el elemento donde ocurrió y luego “sube” por el DOM, activando cualquier manejador de eventos en sus elementos ancestros.

```
<div id="padre">
  <button id="hijo">Haz clic</button>
</div>

<script>
document.getElementById("padre").addEventListener("click", function() {
  alert("Evento en el PADRE (bubbling)");
});

document.getElementById("hijo").addEventListener("click", function() {
  alert("Evento en el HIJO (bubbling)");
});
</script>
```

Si haces clic en el botón *#hijo*, primero se ejecutará su *alert()*, luego el del *#padre*, debido a la propagación ascendente.

En la fase del **capturing**, el evento va desde la raíz del documento hacia el objetivo, antes de ejecutarse en el elemento específico.

Para usar capturing, podemos pasar *{capture: true}* en el *.addEventListener()*.

```
<div id="padre">
  <button id="hijo">Haz clic</button>
</div>

<script>
document.getElementById("padre").addEventListener("click", function() {
  alert("Evento en el PADRE (capturing)");
}, { capture: true });

document.getElementById("hijo").addEventListener("click", function() {
  alert("Evento en el HIJO (capturing)");
});
</script>
```

Ahora, si haces clic en *#hijo*, primero se ejecutará el evento del *#padre*, y luego el del *#hijo*, porque el evento se manejó en la fase de captura.

A veces, no queremos que un evento se propague más allá del elemento en el que ocurrió. Para detenerlo, usamos **`event.stopPropagation()`**.

```
<div id="padre">
  <button id="hijo">Haz clic</button>
</div>

<script>
document.getElementById("padre").addEventListener("click", function() {
  alert("Evento en el PADRE");
});

document.getElementById("hijo").addEventListener("click", function(event) {
  event.stopPropagation(); // 🚫 Evita que el evento burbujee
  alert("Evento en el HIJO");
});
</script>
```

Sin `stopPropagation()`, si clickeas `#hijo`, veras dos alertar: primero el del elemento hijo, luego el elemento padre. Con `stopPropagation()`, solo verás el del `#hijo`, porque el evento no sigue propagándose.

Animaciones en JavaScript

Las animaciones permiten crear efectos visuales dinámicos en una página web. Se pueden lograr de varias maneras, pero las más utilizadas son mediante **`requestAnimationFrame`** y las propiedades de CSS como **`transition`** y **`transform`**.

Animaciones con requestAnimationFrame

El método **`requestAnimationFrame`** permite ejecutar funciones de animación de manera eficiente, sincronizando los fotogramas con la tasa de refresco del monitor y optimizando el rendimiento.


```
function animarElemento(elemento, tiempoInicio) {
  let tiempoActual = performance.now();
  let progreso = (tiempoActual - tiempoInicio) / 1000; // Convertimos a segundos

  if (progreso < 1) {
    elemento.style.transform = `translateX(${progreso * 100}px)`;
    requestAnimationFrame(() => animarElemento(elemento, tiempoInicio));
  } else {
    elemento.style.transform = "translateX(100px)";
  }
}

const elemento = document.getElementById("miElemento");
document.getElementById("iniciar").addEventListener("click", () => {
  requestAnimationFrame((tiempoInicio) => animarElemento(elemento, tiempoInicio));
});
```

1. `requestAnimationFrame` es la clave para hacer que la animación sea suave y eficiente. Se invoca en cada fotograma, lo que permite que la animación se ejecute a una velocidad que se sincroniza con la tasa de refresco de la pantalla.
2. `performance.now()` devuelve el tiempo actual en milisegundos (con una precisión mayor que `Date.now()`), lo cual es útil para medir la duración de la animación.
3. `progreso` se calcula como la diferencia entre el tiempo actual y el tiempo de inicio de la animación, dividido entre 1000 para convertirlo a segundos.
4. `elemento.style.transform = translateX(${progreso * 100}px)`, cada vez que se ejecute el fotograma, la propiedad transform del elemento se actualiza para moverlo en el eje X. El valor de desplazamiento aumenta proporcionalmente con el progreso de la animación.
5. Si `progreso < 1`, sigue ejecutándose la animación. Cuando `progreso >= 1`, se detiene y el elemento se coloca en su posición final `translateX(100px)`.
6. El evento `click` del botón con el ID `iniciar` inicia la animación cuando se hace clic en él.

Transiciones y transformaciones CSS controladas por JavaScript

CSS también ofrece potentes herramientas para crear animaciones sin necesidad de JavaScript, pero a menudo es útil controlar las transiciones y transformaciones desde JavaScript para lograr interactividad y mayor control.

Las **transiciones** CSS permiten animar cambios de propiedades de un elemento de manera suave, pueden ser controladas mediante JavaScript para reaccionar ante eventos o condiciones específicas.

```
const elemento = document.getElementById("miElemento");

// Iniciamos la transición cambiando una propiedad CSS desde JavaScript
function iniciarTransicion() {
    elemento.style.transition = "background-color 0.5s, transform 0.5s";
    elemento.style.backgroundColor = "blue";
    elemento.style.transform = "scale(1.5)";
}

document.getElementById("iniciar").addEventListener("click", iniciarTransicion);
```

1. En este caso, la transición se inicia mediante JavaScript cuando el usuario hace clic en el botón con ID *iniciar*.
2. *elemento.style.transition* especifica las propiedades CSS que se animarán (en este caso, *background-color* y *transform*), con una duración de 0,5 segundos.
3. Luego, se cambian las propiedades del elemento, como el color de fondo y el escalado, lo que activa la animación.

De manera similar a las transiciones, las transformaciones CSS pueden ser controladas y animadas desde JavaScript.

```
const elemento = document.getElementById("miElemento");

function aplicarTransformacion() {
    elemento.style.transition = "transform 0.5s ease";
    elemento.style.transform = "rotate(45deg) scale(1.5)";
}

document.getElementById("iniciar").addEventListener("click", aplicarTransformacion);
```

1. Aquí, al hacer clic en el botón con ID *iniciar*, se aplica una transformación CSS que rota el elemento 45 grados y lo escala al 150% de su tamaño original.
2. *elemento.style.transition* establece que la animación de transformación debe durar 0.5 segundos.
3. La propiedad *transform* se utiliza para cambiar la rotación y el tamaño del elemento.

Drag and Drop

El Drag and Drop es una funcionalidad que permite a los usuarios arrastrar elementos y soltarlos en diferentes ubicaciones dentro de una página web. JavaScript proporciona una API nativa para implementar esta funcionalidad de manera sencilla.

Scope en JavaScript

El scope (o “ambito” en español) en JavaScript define la accesibilidad y el tiempo de vida de las variables dentro del código. Es fundamental para controlar qué partes del programa pueden acceder o modificar ciertas variables.

Tipos de Scope

1. **Scope Global:** las variables declaradas en el scope son accesibles desde cualquier parte del código.

```
let globalVar = "Soy global";

function mostrarGlobal() {
  console.log(globalVar); // "Soy global"
}

mostrarGlobal();
console.log(globalVar); // "Soy global"
```

En este caso, *globalVar* es accesible tanto dentro como fuera de la función.

2. **Scope de Función:** las variables declaradas dentro de una función solo pueden ser accedidas dentro de esa función.

```
function miFuncion() {
  let funcionVar = "Estoy dentro de la función";
  console.log(funcionVar);
}

miFuncion(); // "Estoy dentro de la función"
console.log(funcionVar); // Error: funcionVar is not defined
```

Aquí, *funcionVar* solo existe dentro de *miFunción*, por lo que intentar acceder a ella desde afuera genera un error.

3. **Scope de bloque:** con *let* y *const*, las variables declaradas dentro de un bloque ({ }) solo existen dentro de ese bloque.

```
if (true) {
  let bloqueVar = "Estoy dentro de un bloque";
  console.log(bloqueVar); // "Estoy dentro de un bloque"
}

console.log(bloqueVar); // Error: bloqueVar is not defined
```

Para constantes, el comportamiento es el mismo.

```
if (true) {  
  const bloqueConst = "Soy una constante en un bloque";  
  console.log(bloqueConst); // "Soy una constante en un bloque"  
}  
  
console.log(bloqueConst); // Error: bloqueConst is not defined
```

4. **Scope chain:** define cómo se resuelven las variables cuando se hace referencia a ellas en diferentes niveles de anidación.

Cuando se accede a una variable dentro de una función, JavaScript primero busca la variable en el mismo ámbito local. Si no la encuentra, sube el nivel en la cadena de ámbitos hasta encontrarla en un ámbito superior o, en última instancia, en el scope global.

Desuso de var y preferencia por let y const

Historicamente, `var` fue la única forma de declarar variables en JavaScript, pero tiene varios problemas:

- No respeta el scope de bloque: una variable declarada con `var` dentro de un bloque `if` o `for` sigue siendo accesible fuera de ese bloque.
- Hoisting inesperado: las variables declaradas con `var` se elevan (hoisted) al inicio de su contexto sin inicializarse, lo que puede causar errores difíciles de rastrear.
- Reasignación no controlada: `var` permite declarar la misma variable más de una vez dentro del mismo contexto, lo que puede llevar a sobrescribir valores sin advertencias.

Se recomienda utilizar `let` cuando necesites reasignar el valor de una variable. Usa `const` cuando el valor no deba cambiar.

Control de tiempo en JavaScript

JavaScript permite ejecutar código después de cierto tiempo o de manera repetida utilizando funciones específicas. Estas funciones son especialmente útiles para animaciones, temporizadores, tareas repetitivas, entre otros.

setTimeout()

Esta función se usa para **ejecutar una función una única vez después de un cierto tiempo** (en milisegundos).

```
setTimeout(() => {  
  console.log("Hola después de 2 segundos");  
, 2000);
```

Esta función tiene dos parámetros. El primer parámetro es una función callback que quieras ejecutar y el segundo parámetro es el tiempo de espera en milisegundos (1000ms = 1 segundo).

setInterval()

Se usa para **ejecutar una función de forma repetitiva cada cierto intervalo** de tiempo.

```
const intervalo = setInterval(() => {  
  console.log("Esto se repite cada 1 segundo");  
, 1000);
```

Esta función tiene dos parámetros. Una función callback y el intervalo de tiempo en milisegundos.

clearTimeout()

Sirve para **cancelar la ejecución programada con setTimeout** antes de que ocurra.

```
const temporizador = setTimeout(() => {  
  console.log("Esto no se mostrará");  
, 3000);  
  
clearTimeout(temporizador);
```

clearTimeout() detiene la ejecución si aún no ha ocurrido. Necesitas guardar el ID que retorna *setTimeout()* para poder cancelarlo.

clearInterval()

Similar al anterior, pero **sirve para detener un setInterval()**.

```
const intervalo = setInterval(() => {
  console.log("Esto se repetirá hasta que lo detengamos");
}, 1000);

setTimeout(() => {
  clearInterval(intervalo);
  console.log("Intervalo detenido");
}, 5000);
```

En este ejemplo, el mensaje se repite cada segundo, pero se detiene después de 5 segundos.

Usar `setInterval()` sin un `clearInterval()` puede hacer que tu script siga corriendo indefinidamente. Siempre es buena práctica establecer una condición para detenerlo.

Formularios

Los formularios permiten recopilar datos del usuario. Aunque se construyen con HTML, JavaScript juega un papel crucial para interactuar, validar y procesar la información antes de enviarla al servidor.

Un formulario es una estructura HTML que agrupa campos de entrada como textos, emails, contraseñas, checkboxes, etc., y que normalmente se envía a un servidor para procesar los datos.

```
<form id="miFormulario">
  <label for="nombre">Nombre:</label>
  <input type="text" id="nombre" name="nombre" />

  <label for="edad">Edad:</label>
  <input type="number" id="edad" name="edad" />

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

Interactuar con formularios

Para manejar un formulario con JavaScript, se suelen seguir estos pasos:

1. Capturar el formulario.

```
const formulario = document.getElementById("miFormulario");
```

2. Escuchar el evento `submit`.

```
formulario.addEventListener("submit", function (evento) {
    evento.preventDefault(); // Evita que se recargue la página

    // Aquí va el manejo de los datos
});
```

3. Obtener los valores de los campos.

```
const nombre = document.getElementById("nombre").value;
const edad = document.getElementById("edad").value;

console.log("Nombre:", nombre);
console.log("Edad:", edad);
```

value devuelve el contenido actual del input. *preventDefault()* evita que el formulario se envíe de forma tradicional, permitiendo validarlo y manejarlo con JS.

Validaciones de formularios en el frontend

Validar formularios en el lado del cliente (frontend) es una práctica esencial para:

- Mejorar la experiencia del usuario.
- Evitar enviar datos incorrectos al servidor.
- Ahorrar tráfico y carga innecesaria.

JavaScript permite validar los datos antes de que el formulario sea enviado. Aquí veremos dos formas de hacerlo: con validaciones nativas en HTML5 y con validaciones personalizadas en JavaScript.

El propio **HTML tiene atributos que validan campos** automáticamente.

Atributo	Función
required	Campo obligatorio
type	Valida el tipo (email, number, etc.)
min, max	Valores mínimos y máximos numéricos
minlength, maxlength	Longitud mínima/máxima de texto

Ejemplo:

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

El navegador impide el envío si los datos no cumplen con los requisitos.

Para tener mayor control o mostrar mensajes específicos, podemos validar manualmente a través de JavaScript.

```
const formulario = document.getElementById("formulario");

formulario.addEventListener("submit", function (e) {
  e.preventDefault();

  const nombre = document.getElementById("nombre").value;
  const email = document.getElementById("email").value;

  if (nombre.trim() === "") {
    alert("El nombre es obligatorio");
    return;
  }

  if (!validarEmail(email)) {
    alert("El correo electrónico no es válido");
    return;
  }

  // Si todo es válido
  console.log("Formulario válido. Puedes enviarlo.");
});

function validarEmail(correo) {
  const expresion = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
  return expresion.test(correo);
}
```

Además de `alert()`, podemos mostrar los errores junto al campo:

```
if (nombre.trim() === "") {
  document.getElementById("nombre").classList.add("campo-error");
  document.getElementById("errorNombre").textContent = "Ingresa tu nombre";
}
```

Recomendaciones:

- Combina validaciones HTML5 con JavaScript para mayor seguridad y control.
- Muestra mensajes claros y específicos.
- Marca los campos con errores visualmente.

Peticiones HTTP

Las peticiones HTTP son un conjunto de métodos predefinidos que se utilizan para interactuar con los datos almacenados en los servidores.

Cada solicitud enviada a un servidor incluye un punto final y el tipo de solicitud que se envía. Puedes ver un punto final como una pasarela entre dos programas: el cliente y el servidor.

El **cliente es el programa que envía la solicitud mientras que el servidor es el que recibe la solicitud**. El servidor devuelve una respuesta en función de la validez de la solicitud. Si la solicitud tiene éxito, el servidor devuelve los datos en formato XML o JSON (JSON en la mayoría de los casos), y si la solicitud falla, el servidor devuelve un mensaje de error.

Las respuestas que devuelve el servidor suelen estar asociadas a códigos de estado. Estos códigos nos ayudan a entender lo que el servidor intenta decir cuando recibe una petición.

Métodos HTTP

Las peticiones HTTP se clasifican según su método, que indica que acción se quiere realizar.

Método	¿Qué hace?	Uso común
GET	Obtener datos del servidor	Leer usuarios, productos...
POST	Envía datos al servidor	Formularios, nuevos datos...
PUT	Reemplaza completamente un recurso	Editar algo existente
PATCH	Modifica parcialmente un recurso	Cambios pequeños
DELETE	Elimina un recurso del servidor	Borrar usuarios, tareas...

¿Qué es una API?

Una **API (Application Programming Interface)** es un conjunto de reglas que permite que diferentes aplicaciones se comuniquen entre sí.

En el contexto web, una API expone “puertas” o “rutas” que puedes usar para hacer peticiones HTTP y obtener datos, enviarlos o modificarlos.

Por ejemplo, una API de productos puede tener una ruta `/productos` para obtener la lista completa, y otra `/productos/1` para obtener solo uno.

En resumen:

- HTTP: protocolo que hace posible la comunicación.
- Métodos HTTP: las acciones posibles dentro de esa comunicación.
- API: el intermediario que recibe las peticiones y devuelve los datos.

Peticiones y promesas

Ahora que sabemos qué son las peticiones HTTP, vamos a entrar en las diferencias entre dos tipos de peticiones:

- **Petición síncrona:** las tareas se ejecutarán de forma secuencial. Esto significa que una tarea debe completarse antes de que la siguiente comience. Si una operación toma mucho tiempo (por ejemplo, cargar datos desde un servidor), el navegador esperará a que termine esa operación antes de continuar con las demás tareas. Durante este tiempo, la aplicación se “congela” y no responde a otras interacciones del usuario.
- **Petición asíncrona:** las tareas no esperan a que las operaciones anteriores terminen. En su lugar, el código sigue ejecutándose mientras una operación de larga duración (como una solicitud de datos a una API) se completa en segundo plano. Cuando la operación asíncrona finaliza, se maneja su resultado (éxito o error) sin interrumpir la ejecución del resto del código.

Las peticiones asíncronas toman mucha importancia en JavaScript ya que este, es un lenguaje **single-threaded** (de un solo hilo), lo que significa que solo puede ejecutar una tarea a la vez. Si se realizan muchas tareas síncronas que consumen mucho tiempo (como solicitudes de red o lecturas de archivos grandes), esto puede hacer que la aplicación se vuelva lenta o deje de responder. **Las operaciones asíncronas permiten a JavaScript gestionar tareas de larga duración sin bloquear la ejecución del resto del programa**, mejorando la experiencia del usuario.

Ahora tenemos entender qué es una promesa. **Una promesa es un objeto que representa la eventual finalización o el fracaso de una operación asíncrona.**

Básicamente, es un intermediario entre el código que hace la petición y el código que espera la respuesta.

Estas promesas suelen tener 3 estados:

- **Pending (pendiente):** la operación aún no se ha completado.
- **Fulfilled (cumplida):** la operación se completó con éxito.
- **Rejected (rechazada):** la operación falló.

```
fetch('https://api.example.com/datos')
  .then(respuesta => respuesta.json())
  .then(datos => console.log(datos))
  .catch(error => console.error(error));
```

Conectando con el servidor: `fetch()` y Web APIs esenciales

Las aplicaciones web modernas no funcionan de manera aislada. Casi todas necesitan comunicarse con servidores para obtener o enviar datos. JavaScript nos permite hacerlo gracias a herramientas como *fetch()*, junto con Web APIs del navegador como *localStorage* y *sessionStorage*.

fetch() es una función nativa de JavaScript que permite hacer peticiones HTTP de manera asíncrona. Es la evolución de *XMLHttpRequest*, y proporciona una forma más limpia y moderna de interactuar con APIs.

Con *fetch()* puedes:

- Obtener datos (GET)
- Enviar datos (POST, PUT, PATCH)
- Eliminar recursos (DELETE)
- Configurar encabezados, cuerpo y otros detalles.

Sintaxis básica de `fetch()`

```
fetch(url, opciones)
  .then(respuesta => {
    // Procesar la respuesta
  })
  .catch(error => {
    // Manejo de errores
  });
```

- *url*: es la dirección del recurso al que quieres acceder, que pueden ser una API REST, un archivo, o cualquier recurso disponible en línea.
- *opciones*: es un objeto opcional que puedes usar para configurar aspectos de la solicitud, como el método (GET, POST, PUT, DELETE), los encabezados, el cuerpo de la solicitud, entre otros.

La función *fetch()* **devuelve una promesa** que será aceptada cuando reciba una respuesta y sólo será rechazada si hay un fallo de red o si por alguna razón no se pudo completar la petición.

El modo más habitual de manejar las promesas es utilizando `.then()`, aunque también se puede utilizar `async/await`.

Al método `.then()` se le pasa una función callback donde su parámetro respuesta (res o response también) es el objeto de respuesta de la petición que hemos realizado. En su interior realizaremos la lógica que queramos hacer con la respuesta a nuestra petición.

Manejo de errores con `fetch()`

Es importante entender que `fetch()` solo rechaza la promesa en caso de un **error de red** (por ejemplo, si no puede alcanzar el servidor). Sin embargo, si el servidor responde con un código de estado como **404** (No encontrado) o **500** (Error interno del servidor), `fetch()` **no** rechazará la promesa. En cambio, la promesa se resolverá con una respuesta que contiene el código de estado correspondiente, aunque no sea una respuesta exitosa.

Por lo tanto, es responsabilidad del programador verificar manualmente si la respuesta es exitosa. Esto se puede hacer comprobando la propiedad `response.ok`, que será `true` si el código de estado está en el rango **200** (lo que indica una respuesta exitosa), o `false` en caso contrario.

```
fetch('https://api.ejemplo.com/datos')
.then(response => {
  // Verificamos si la respuesta fue exitosa (código 2xx)
  if (!response.ok) {
    // Si la respuesta no es exitosa, lanzamos un error
    throw new Error('Error al obtener los datos: ' + response.status);
  }
  return response.json(); // Si la respuesta es exitosa, parseamos el JSON
})
.then(data => {
  console.log('Datos recibidos:', data);
})
.catch(error => {
  // Capturamos cualquier error de red o error lanzado manualmente
  console.error('Error:', error);
});
```

1. Se realiza una solicitud `fetch()` a la API.
2. Dentro del primer `.then()`, se verifica la propiedad `response.ok`. Si es `false`, se lanza un error con el código de estado (como `404` o `500`).
3. Si la respuesta es exitosa (código 200), se convierte el cuerpo de la respuesta a JSON con `response.json()`.
4. En el segundo `.then()`, se manejan los datos si la solicitud fue exitosa.
5. Si ocurre un error en cualquier parte del proceso (ya sea un error de red o un error lanzado manualmente), se captura con `.catch()`.

Configuración de headers

Los headers (cabeceras HTTP) **son pares clave-valor que se envían junto con una solicitud o una respuesta HTTP**. Informan al servidor sobre el contenido, el tipo de datos, el formato esperado, autenticación, etc.

Por ejemplo:

- Qué tipo de contenido estás enviando (Content-Type)
- Qué formato de respuesta esperas (Accept)
- Información de autenticación (Authorization)
- Control de comportamiento del caché (Cache-Control)

Estos headers **se configuran dentro del objeto de opciones que pasas a fetch** como segundo parámetro:

```
fetch('https://api.example.com/data', {
  method: 'POST', // Puede ser GET, POST, PUT, DELETE, etc.
  headers: {
    'Content-Type': 'application/json',
    'Accept': 'application/json',
    'Authorization': 'Bearer token_aquí'
  },
  body: JSON.stringify({ nombre: 'Juan', edad: 30 })
})
.then(response => response.json())
.then(data => console.log(data))
.catch(error => console.error('Error:', error));
```

1. **‘Content-Type’: ‘application/json’** Indica que los datos que estás enviando en el body están en formato JSON. El servidor necesita saber cómo interpretar lo que le estás mandando. En este ejemplo le estamos enviando el objeto `{“nombre”: “Juan”, “edad”: 30}`.
2. **‘Accept’: ‘application/json’** Le comunicas al servidor que esperas recibir una respuesta en formato JSON. Algunos servidores pueden responder en distintos formatos (HTML, XML, JSON).
3. **‘Authorization’: ‘Bearer token_aquí’** Este header se usa para autenticación. El token (cadena de texto) permite identificar que tienes permiso para acceder a esa API. Bearer es un tipo de autenticación. Le dice al servidor que estás usando un “token portador”.

Códigos de estado importantes para tener en cuenta

- 100-199: respuestas informativas
- 200-299: respuestas exitosas

- 300-399: redirecciones
- 400-499: errores del cliente
- 500-599: errores del servidor

Web APIs esenciales

Las Web APIs son interfaces proporcionadas por los navegadores que permiten a las aplicaciones web interactuar con diferentes aspectos del navegador o del sistema operativo.

localStorage y sessionStorage

Son dos Web APIs que permiten almacenar datos en el navegador del usuario. La principal diferencia entre ellas radica en el tiempo de vida de los datos:

- **localStorage:** permite almacenar datos de manera persistente, es decir, los datos se mantienen incluso cuando el navegador se cierra o la computadora se reinicia. Los datos solo se eliminan si el usuario borra el caché del navegador o utiliza métodos específicos para eliminar los datos.
- **sessionStorage:** almacena datos solo durante la sesión de navegación. Los datos se eliminan cuando el navegador o la pestaña se cierran.

```
// Guardar un dato en localStorage o sessionStorage
localStorage.setItem('nombre', 'Juan');
sessionStorage.setItem('token', 'abc123');

// Recuperar el dato de localStorage o sessionStorage
let nombre = localStorage.getItem('nombre');
let token = sessionStorage.getItem('token');
```

WebSockets

Son una **tecnología que permite una comunicación bidireccional en tiempo real entre navegador y el servidor**. A diferencia de las peticiones HTTP, que son unidireccionales, WebSockets mantienen una conexión abierta entre cliente y el servidor, lo que permite intercambiar datos de manera eficiente y en tiempo real. Esta tecnología es ideal para aplicaciones que requieren actualizaciones constantes y rápidas, como chats, juegos multijugador y notificaciones en tiempo real.

Service Workers (caché y notificaciones push)

Los Service Workers **son scripts que el navegador ejecuta en segundo plano, independientemente de la página web que esté abierta**. Esto permite realizar tareas como caché de recursos (para que las aplicaciones web funcionen sin conexión) y notificaciones push (para alertar a los usuarios de eventos importantes incluso cuando no están en la página).

Estos Service Workers pueden interceptar las peticiones de red y almacenar los recursos en un caché, lo que permite que la aplicación siga funcionando incluso sin conexión a internet.

Instalación del SW:

```
// Instalación del Service Worker
self.addEventListener('install', (event) => {
  event.waitUntil(
    caches.open('mi-cache').then((cache) => {
      return cache.addAll([
        '/index.html',
        '/style.css',
        '/app.js'
      ]);
    })
  );
});
```

- `self` es una referencia al Service Worker en sí mismo.
- Se está escuchando el evento `'install'`, que ocurre cuando el Service Worker se instala por primera vez.
- `event.waitUntil(...)` le dice al navegador que espere hasta que se complete la promesa que le pases antes de marcar el Service Worker como instalado correctamente.
- Si esta promesa falla, la instalación de SW también falla.
- Se abre (o se crea si no existe) una caché llamada `'mi-cache'`.
- `cache.addAll([...])` guarda esos archivos en la caché. Eso significa que después estarán disponibles, aunque el usuario esté offline.

Con esto estás precargando archivos importantes de tu app en la caché desde el momento en que se instala el SW. Así, si el usuario entra sin conexión, aún puede acceder a `/index.html`, `/style.css` y `/app.js`.

```
// Intercepción de las solicitudes de red
self.addEventListener('fetch', (event) => {
  event.respondWith(
    caches.match(event.request).then((cachedResponse) => {
      // Si el recurso está en caché, lo devolvemos
      if (cachedResponse) {
        return cachedResponse;
      }
      // Si no está en caché, lo solicitamos desde la red
      return fetch(event.request);
    })
  );
});
```

- Este código (el evento `fetch`) se dispara cada vez que la página hace una solicitud HTTP.
- `self` otra vez se refiere al SW.
- `respondWith()` toma una promesa que debe devolver un objeto `Response` (la respuesta que irá de vuelta al navegador).
- `caches.match()` busca si ya hay una versión cacheada del recurso que se está pidiendo (`event.request`).
- Si existe en caché, se devuelve esa versión inmediatamente. Si no está cacheado, entonces se hace una solicitud normal a la red (`fetch`), y esa respuesta se usará.

Los SW también pueden recibir y mostrar notificaciones push aunque la aplicación web no esté activa, lo que las hace útiles para enviar actualizaciones a los usuarios.

```
// Registramos el Service Worker
if ('serviceWorker' in navigator && 'PushManager' in window) {
  navigator.serviceWorker.register('/service-worker.js')
    .then((registration) => {
      console.log('Service Worker registrado con éxito:', registration);
    })
    .catch((error) => {
      console.error('Error al registrar el Service Worker:', error);
    });
}
```

Este bloque registra tu SW y verifica si el navegador soporta tanto los SW como las notificaciones push.

- `'serviceWorker' in navigator` comprueba si el navegador permite registrar un SW.
- `'pushManager' in window` comprueba si puede manejar notificaciones push.
- Llama al archivo `/service-worker.js` y lo registra como el SW de tu sitio.
- Este archivo debe estar en la raíz del sitio o en un scope apropiado, para que pueda controlar todo el contenido.
- `registration` contiene información sobre el SW (como su estado, etc.)

- Si ocurre un error durante el registro (por ejemplo, el archivo no se encuentra), se captura aquí.

Este bloque le pregunta al usuario si permite que el sitio le envíe **notificaciones push**, que pueden aparecer incluso si la página está cerrada.

```
// Solicitar permiso para recibir notificaciones push
Notification.requestPermission().then((permission) => {
  if (permission === 'granted') {
    console.log('Permiso concedido para recibir notificaciones push');
  }
});
```

- Llama a la función `Notification.requestPermission()`, que lanza una ventana emergente del navegador.
- El usuario elige entre: 'granted' (permitido), 'denied' (denegado) o 'default' (cerro sin elegir nada).
- Devuelve una promesa que se resuelve con esa elección.
- Si el usuario acepta, se mostrará mensaje por consola. En este punto, ya podrías enviar notificaciones usando un SW y la API Push.

Patrones de diseño

Module Pattern (Patrón Módulo)

El **patrón módulo** es una técnica de diseño que se utiliza para encapsular código y crear funciones o variables “privadas” y “públicas”.

Este patrón de diseño se basa en el uso de funciones autoejecutables (IIFE – Immediately Invoked Function Expressions). Este tipo de función crea un ámbito (scope) cerrado, dentro del cual **puedes definir variables y funciones privadas**, y luego exponer solo lo que quieras como público.

```
const miModulo = (function() {
  // Variables y funciones privadas
  let contador = 0;

  function incrementar() {
    contador++;
    console.log(contador);
  }

  // Exponer solo lo que quieres que sea público
  return {
    incrementarContador: incrementar
  };
})();
```

En el anterior ejemplo, *contador* e *incrementar* no son accesibles directamente desde fuera. Solo *incrementarContador* es público y accesible desde el exterior.

Con la llegada de ES6 y los módulos nativos (import/export), el patrón módulo clásico ha perdido un poco de uso, pero sigue siendo muy útil en ciertos contextos o en código legacy.

Factory Pattern (Patrón Fábrica)

El **patrón fábrica** es un patrón de diseño creacional que **se usa para crear objetos sin especificar la clase exacta del objeto que se va a crear**. En lugar de usar *new* directamente, se delega la creación de objetos a una función o método “fabrica”.

```
function crearPersona(tipo) {
  if (tipo === 'estudiante') {
    return {
      tipo: 'Estudiante',
      estudiar: function() {
        console.log('Estudiando...');
      }
    };
  } else if (tipo === 'profesor') {
    return {
      tipo: 'Profesor',
      enseñar: function() {
        console.log('Enseñando...');
      }
    };
  } else {
    return {
      tipo: 'Desconocido'
    };
  }
}

// Uso
const persona1 = crearPersona('estudiante');
persona1.estudiar(); // Estudiando...

const persona2 = crearPersona('profesor');
persona2.enseñar(); // Enseñando...
```

¿Cuándo usarlo?

- Cuando el proceso de creación es complejo.
- Cuando quieres devolver subtipos o diferentes objetos basados en una condición.

Observer Pattern (Patrón Observador)

El **patrón observador** es un patrón de diseño de comportamiento que permite que un objeto (el “sujeto”) notifique a otros objetos (los “observadores”) cuando **cambia su estado**, sin que estén fuertemente acoplados.

Es como el sistema de suscripción/notificación: los observadores se “suscriben” a un sujeto, y cuando algo cambia, el sujeto les avisa automáticamente.

```
// Sujeto (Subject)
class Sujeto {
  constructor() {
    this.observadores = [];
  }

  agregarObservador(observador) {
    this.observadores.push(observador);
  }

  eliminarObservador(observador) {
    this.observadores = this.observadores.filter(obs => obs !== observador);
  }

  notificar(data) {
    this.observadores.forEach(obs => obs.actualizar(data));
  }
}
```

```
// Observador (Observer)
class Observador {
  constructor(nombre) {
    this.nombre = nombre;
  }

  actualizar(data) {
    console.log(`${this.nombre} recibió: ${data}`);
  }
}
```

```
// Uso
const sujeto = new Sujeto();

const obs1 = new Observador("Observer 1");
const obs2 = new Observador("Observer 2");

sujeto.agregarObservador(obs1);
sujeto.agregarObservador(obs2);

sujeto.notificar("Nuevo mensaje");
// Output:
// Observer 1 recibió: Nuevo mensaje
// Observer 2 recibió: Nuevo mensaje
```

