ASINCRONISMO EN JAVASCRIPT

TEMA 01 – INTRODUCCIÓN AL ASINCRONISMO EN JAVASCRIPT

1.1. Introducción al asincronismo en JavaScript

Una de sus características más relevantes de JavaScript es su modelo de ejecución asíncrono, conocido por permitir que múltiples tareas se gestionen sin bloquear la aplicación mientras se esperan respuestas u operaciones de larga duración.

¿Qué significa “asincronismo” en el contexto de la programación?

El término **asincronismo** hace referencia a la capacidad de un programa de ejecutar ciertas **tareas** en **segundo plano**, permitiendo que el **flujo principal** continúe con la ejecución de otras instrucciones **sin** tener que **detenerse** hasta que se complete la tarea anterior. En un lenguaje de programación tradicionalmente imperativo y secuencial, como podría ser C o Java en su forma más básica, las sentencias se ejecutan de manera **síncrona**: la ejecución de una instrucción depende de que la anterior haya finalizado. Esto implica que, en presencia de operaciones de **E/S** (Entrada/Salida) o conexiones de red que requieren mucho tiempo, el **programa** queda **bloqueado** hasta obtener los resultados.

JavaScript, en cambio, adopta un enfoque diferente: **no se queda bloqueado mientras se espera la finalización de operaciones costosas** (por ejemplo, llamadas a una API externa, lecturas de archivos de disco, consultas a bases de datos, etc.). En lugar de ello, se apoya en un mecanismo de gestión de tareas que **permite registrar** las **operaciones asíncronas** y **continuar** con la ejecución del código **sin bloqueos**. Al completarse dichas operaciones, se activa una función de devolución de llamada (**callback**), o se resuelve una Promesa (**Promise**), posibilitando el

procesamiento de los resultados cuando están disponibles, sin haber entorpecido el rendimiento del resto de la aplicación.

Este modelo de asincronismo es clave para el desarrollo de aplicaciones web modernas que buscan ofrecer una experiencia de usuario fluida, evitando bloqueos o tiempos de espera prolongados que puedan frustrar la interacción del usuario con la aplicación. En este sentido, JavaScript se ha posicionado como un referente en la creación de interfaces altamente dinámicas y reactivas, donde el asincronismo no solo es una ventaja, sino prácticamente una necesidad.

1.2. Naturaleza de un solo hilo en JavaScript

Un aspecto fundamental para comprender el asincronismo en JavaScript es su naturaleza de un solo hilo (**single-threaded**). En términos generales, un hilo es la secuencia de ejecución de instrucciones dentro de un proceso. Un lenguaje single-threaded dispone de un único hilo principal encargado de ejecutar el código. Esto significa que el intérprete de JavaScript procesa las líneas de código de arriba hacia abajo en un solo flujo.

A diferencia de otros lenguajes o entornos que pueden utilizar múltiples hilos para paralelizar tareas, **JavaScript ejecuta todo su código en un único hilo**. Sin embargo, esto no le impide manejar varias operaciones de manera simultánea, al menos desde la perspectiva del desarrollador. ¿Cómo lo consigue? La clave reside en el **Event Loop** (bucle de eventos).

Event Loop

El **Event Loop** es el **mecanismo interno** que permite coordinar las tareas asíncronas en JavaScript. Este bucle recibe y **gestiona los eventos y tareas pendientes de ser procesados**, uno tras otro, reanudando la ejecución del flujo principal únicamente cuando está listo para ello. En términos simplificados:

1. Se mantiene una **cola de tareas** en la que se registran todos los **callbacks**, **promesas** u **operaciones asíncronas** que están **pendientes**.

2. **JavaScript ejecuta, en primer lugar, el código principal de manera síncrona**, línea a línea, hasta que se completa o encuentra una operación asíncrona.

3. Cuando se desencadena una **operación asíncrona** (por ejemplo, una petición HTTP), esta **se delegará** al entorno externo (como las APIs del navegador), para que se procese.

4. Mientras tanto, **el flujo principal no se bloquea**: continúa ejecutando las siguientes líneas de código.

5. Una vez **finalizada la operación asíncrona**, **se añade a la cola de tareas**. El **Event Loop comprueba si el hilo principal está libre** y, cuando lo está, **retoma la tarea pendiente**, ejecutando el callback o resolviendo la Promesa.

Gracias a este proceso, JavaScript puede resultar muy eficiente al coordinar tareas que involucran E/S o consultas remotas sin que la aplicación quede bloqueada.

1.3. Callbacks, promesas y async/await

A continuación, se presenta una breve introducción a los callbacks, las promesas y la sintaxis async/await en JavaScript, centrada en la importancia de cada uno de estos elementos dentro de la programación asíncrona. Dado que estamos en una etapa inicial dentro del asincronismo en JavaScript, no se profundizará ahora en ninguno de estos aspectos, dejando esta tarea para más adelante, en documentos dedicados.

Callbacks

Un *callback* (o función de retorno) es una función que se pasa como parámetro a otra función y se ejecuta una vez que la tarea principal ha concluido. Históricamente, los callbacks han sido la forma más habitual de manejar la asincronía en JavaScript, especialmente antes de que existieran las promesas y la sintaxis async/await.

Ventajas

• Permiten seguir ejecutando código mientras se espera la respuesta de una operación (por ejemplo, una petición a una API externa).

• Son relativamente sencillos de utilizar para operaciones pequeñas.

Desventajas

• Cuando se encadenan varios callbacks (por ejemplo, varias llamadas asíncronas consecutivas), puede aparecer lo que se conoce como *callback hell*, dificultando la lectura y mantenimiento del código.

Ejemplo básico de callback

Imagina que deseas simular la obtención de datos desde un servidor, y una vez que los tienes, imprimes esos datos en la consola.

```
function obtenerDatos(accionAlTerminar) {
  console.log("Obteniendo datos del servidor...");
  // Simulamos el retardo en la obtención de datos con setTimeout
  setTimeout(() => {
    const datos = { nombre: "Alumno", curso: "Desarrollo Web" };
    // Una vez obtenidos los datos, ejecutamos el callback
    accionAlTerminar(datos);
  }, 2000);
}
function imprimirDatos(datos) {
  console.log("Datos recibidos:", datos);
}
// Llamamos a la función principal y le pasamos el callback
obtenerDatos(imprimirDatos);
```

En este ejemplo, la función obtenerDatos recibe una función de callback llamada accionAlTerminar que se ejecuta cuando los datos ya están

listos. Este método es funcional pero, como verás más adelante, puede resultar poco legible cuando se tienen muchas tareas encadenadas.

2. Promesas (Promises)

Las *promesas* llegaron para aportar una forma más clara y organizada de manejar la asincronía. Una promesa es un objeto que representa la finalización exitosa o el fracaso de una operación asíncrona. Existen tres estados básicos en una promesa:

1. ***Pending*** (pendiente): la promesa está en proceso de ser resuelta.
2. ***Fulfilled*** (cumplida): la operación terminó satisfactoriamente.
3. ***Rejected*** (rechazada): la operación falló.

Ventajas

• Ofrecen una sintaxis más clara respecto a los callbacks anidados.

• Facilitan el manejo de errores, gracias a los métodos `then()` y `catch()`.

• Permiten encadenar múltiples operaciones asíncronas sin caer en el *callback hell*.

Ejemplo básico con promesas

En el siguiente ejemplo, convertimos la simulación de obtener datos en una función que retorna una promesa:

```
function obtenerDatosPromesa() {
  console.log("Obteniendo datos del servidor...");
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      const datos = { nombre: "Alumno", curso: "Desarrollo Web" };
      // Simulamos un escenario de éxito
      if (datos) {
        resolve(datos);  // La promesa se cumple
```

```
      } else {
          reject("Error al obtener los datos"); // La promesa se
rechaza
      }
    }, 2000);
  });
}
// Uso de la promesa
obtenerDatosPromesa()
  .then((respuesta) => {
    console.log("Datos recibidos:", respuesta);
  })
  .catch((error) => {
    console.error("Ha ocurrido un error:", error);
  });
```

Aquí, la función *`obtenerDatosPromesa`* retorna una promesa. Si todo va bien, se llama a *`resolve(datos)`* y la promesa pasa a estado fulfilled. Si algo falla, se llama a *`reject(...)`*, y la promesa pasa a estado rejected, ejecutando el método *`catch()`*.

Async/Await

La sintaxis *async/await* es una **forma más reciente** e intuitiva de manejar las operaciones asíncronas. Básicamente, *`async`* **marca una función como asíncrona**, y *`await`* **hace que la ejecución se detenga hasta que la promesa se resuelva o se rechace,** permitiéndonos escribir código que se lee como si fuese sincrónico.

Ventajas

- Código más legible y fácil de seguir.
- Mantenimiento simplificado, sobre todo en proyectos grandes.

• Mejor manejo de errores gracias al uso de try/catch dentro de funciones asíncronas.

Ejemplo básico con async/await

Tomando la misma función *obtenerDatosPromesa*, podemos consumirla usando async/await:

```
// Función que retorna una promesa
function obtenerDatosPromesa() {
  console.log("Obteniendo datos del servidor...");
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      const datos = { nombre: "Alumno", curso: "Desarrollo Web" };
      // Simulamos un escenario de éxito
      if (datos) {
        resolve(datos); // La promesa se cumple
      } else {
          reject("Error al obtener los datos"); // La promesa se
rechaza
      }
    }, 2000);
  });
}
// Función asíncrona que utiliza la promesa
async function mostrarDatos() {
  try {
    console.log("Llamando a la función obtenerDatosPromesa...");
    const respuesta = await obtenerDatosPromesa();
    console.log("Datos recibidos:", respuesta);
  } catch (error) {
    console.error("Ha ocurrido un error:", error);
  }
}
```

```
// Llamada a la función asíncrona
mostrarDatos();
```

En este ejemplo, la palabra clave *`await`* pausa la ejecución hasta que la promesa se resuelva o rechace. Si la promesa se cumple, la variable respuesta obtiene el valor devuelto por *`resolve(datos)`*. Si se produce un error, se maneja en el bloque catch del *`try/catch`*.

Conclusión

1. **Callbacks**: fueron la primera herramienta para manejar la asincronía; funcionan bien en situaciones simples, pero pueden producir problemas de legibilidad con operaciones más complejas.

2. **Promesas**: ofrecen un mejor control de flujo y manejo de errores que los callbacks. Presentan una forma más clara de encadenar varias operaciones asíncronas.

3. **Async/Await**: mejora la legibilidad y el control de los flujos asíncronos, haciendo que el código se aproxime mucho al estilo sincrónico, aunque por debajo siguen funcionando con promesas.

1.4. Ejemplos

A continuación se presentan ejemplos finales de los conceptos tratados en este documento introductorio.

Ejemplo 1: Saludo retardado con setTimeout

Este primer ejemplo muestra cómo una función se ejecuta de forma diferida después de un intervalo de tiempo, sin detener el flujo principal del programa.

```
console.log("Mensaje 1: Inicio del programa.");

setTimeout(function() {
  console.log("Mensaje 2: Este saludo aparece tras 2 segundos.");
}, 2000);
```

```
console.log("Mensaje 3: Fin del programa (se ejecuta antes del saludo retrasado).");
```

1. El intérprete de JavaScript ejecuta la línea *`console.log("Mensaje 1...");`* inmediatamente y muestra *“Mensaje 1: Inicio del programa.”*.
2. Encuentra la función *`setTimeout(...)`*. Ésta agenda la ejecución de la función callback dentro de 2 segundos, pero **no bloquea** la ejecución de las siguientes líneas.
3. Mientras pasan esos 2 segundos, el hilo principal continúa, y se muestra *“Mensaje 3: Fin del programa (se ejecuta antes del saludo retrasado).”*.
4. Una vez transcurridos los 2 segundos, el **Event Loop** inserta la tarea pendiente en la cola de eventos, y finalmente se ejecuta la función callback, mostrando *“Mensaje 2: Este saludo aparece tras 2 segundos.”*.

Con ello, se ilustra cómo JavaScript, pese a trabajar con un único hilo, permite coordinar tareas de forma asíncrona sin detener el flujo de ejecución.

Ejemplo 2: Cálculo en segundo plano con setTimeout y funciones callback

Aunque JavaScript es single-thread, a menudo necesitamos realizar operaciones de cierta complejidad (por ejemplo, cálculos aritméticos intensivos) sin bloquear la interfaz de usuario o el flujo principal. En este ejemplo, se simula un cálculo intensivo usando *`setTimeout`* para pasar la tarea a segundo plano, permitiendo a la aplicación seguir funcionando.

```
function calcularFactorial(num, callback) {
  // Función recursiva para calcular el factorial
  function factorial(n) {
    if (n <= 1) return 1;
    return n * factorial(n - 1);
  }
```

```
  console.log("Iniciando cálculo factorial de", num);
  // Simula una tarea compleja en segundo plano
  setTimeout(function() {
    const resultado = factorial(num);
    // Una vez calculado, se ejecuta el callback con el resultado
    callback(resultado);
  }, 0);
}
// Uso de la función 'calcularFactorial'
calcularFactorial(10, function(resultado) {
  console.log("El factorial es:", resultado);
});
// Mientras tanto, el programa continúa...
console.log("El programa sigue ejecutándose sin esperar el cálculo...");
```

1. La función *`calcularFactorial`* recibe un número y un callback.
2. Dentro de *`calcularFactorial`*, se realiza la llamada a *`setTimeout`* con un tiempo de retardo de 0 milisegundos. Aunque parezca que es “inmediato”, en realidad el callback de setTimeout se **agenda** para la siguiente iteración del Event Loop, permitiendo a JavaScript continuar ejecutando otras tareas mientras tanto.
3. El hilo principal registra la llamada a *`setTimeout`* y continúa con la ejecución, por lo que se muestra en consola el mensaje *`“El programa sigue ejecutándose…”`*.
4. Cuando el Event Loop detecta que ya puede atender la tarea pendiente, ejecuta la función callback que calcula el factorial y muestra el resultado en la consola.

Este patrón es muy útil para fraccionar operaciones de alto coste computacional, evitando “congelar” la interfaz o bloquear otros procesos durante la ejecución.

Ejemplo 3: Obteniendo datos de una API con fetch

En este último ejemplo, se muestra cómo realizar una petición HTTP asíncrona para obtener datos de un servicio externo (una API), utilizando la función `fetch()`. Se utiliza la API pública de JSON Placeholder para demostrar la operación.

```
// URL de la API pública
const apiURL = "https://jsonplaceholder.typicode.com/posts/1";

console.log("Iniciando petición a la API...");

fetch(apiURL)
  .then(function (response) {
    // Se recibe la respuesta y se convierte a formato JSON
    return response.json();
  })
  .then(function (data) {
    // Cuando se dispone de los datos en formato objeto JS, se muestran en consola
    console.log("Datos recibidos:", data);
  })
  .catch(function (error) {
    // En caso de error en la conexión u otro problema, se captura aquí
    console.error("Error al obtener los datos:", error);
  });

console.log("La petición se ha iniciado, el programa sigue con otras tareas...");
```

1. La llamada a `fetch(apiURL)` se ejecuta de inmediato; sin embargo, **no se bloquea** el hilo principal mientras se espera la respuesta del servidor.

2. El método `then()` se encarga de procesar la respuesta cuando llegue. JavaScript almacena internamente una promesa que se resolverá al completar la petición.

3. Mientras tanto, el resto del código sigue ejecutándose, de modo que el mensaje `"La petición se ha iniciado…"` aparece inmediatamente en la consola, **antes** de que los datos de la API estén disponibles.

4. Cuando el servidor devuelve la respuesta, el Event Loop programa la ejecución de la función callback correspondiente en la cola de tareas. Con esto, finalmente se llama a `then(function(data) {...})`, donde ya se manejan los datos devueltos.

5. Si ocurre algún error (fallo de red, URL incorrecta, etc.), el flujo salta al catch, capturando así la excepción de la promesa rechazada.

Este ejemplo demuestra de forma muy clara cómo se gestiona de manera asíncrona la comunicación con servicios externos.

Conclusiones

Estos tres ejemplos ilustran cómo JavaScript gestiona de forma asíncrona operaciones que tardan un tiempo en completarse (esperar un intervalo de tiempo, recibir datos de una API o efectuar un cálculo costoso). Observa que en todos los casos:

- La ejecución principal no se detiene.
- El **Event Loop** se encarga de reasignar las tareas pendientes una vez que estén listas.
- Se usan mecanismos como **callbacks**, **promesas** y funciones como **setTimeout** y **fetch** para coordinar el trabajo en segundo plano.

ASINCRONISMO EN JAVASCRIPT

TEMA 02 – BLOQUEO VS NO BLOQUEO

2.1. Introducción al concepto de bloqueo

En el entorno de JavaScript, tanto en el lado del cliente (navegadores web) como en el lado del servidor (Node.js), es fundamental comprender las diferencias entre las operaciones que bloquean la ejecución y aquellas que no lo hacen. Este conocimiento resulta clave para desarrollar aplicaciones eficientes, escalables y con un rendimiento adecuado. A continuación, profundizaremos en los conceptos de sincronía/asíncronía y presentaremos ejemplos para ilustrar el impacto del bloqueo y la importancia de manejar correctamente las operaciones que pueden afectar al rendimiento de nuestras aplicaciones.

2.2. Diferencia entre operaciones síncronas y asíncronas

¿Qué se entiende por operaciones síncronas?

Las operaciones **síncronas** se llevan a cabo **secuencialmente**. Cuando el programa se encuentra con una operación síncrona, detiene su ejecución hasta que dicha operación ha concluido. Solo entonces pasa a la instrucción siguiente. En términos sencillos, el flujo del programa espera la finalización de una tarea antes de continuar.

Ventajas de las operaciones síncronas

• **Simplicidad**: El flujo de trabajo es más fácil de razonar, ya que cada instrucción se ejecuta en orden estricto y no hay “saltos” temporales.

• **Código lineal**: Debido a que no se delega nada a rutinas asíncronas, el código suele escribirse de manera más directa (un paso detrás de otro), facilitando la lectura en escenarios muy simples.

Desventajas de las operaciones síncronas

• **Bloqueo de la ejecución**: Si la operación lleva demasiado tiempo (por ejemplo, cálculos intensivos o acceso a recursos externos con

latencia, como consultas a bases de datos o peticiones de red), el hilo de ejecución principal permanecerá bloqueado. Esto implica que la aplicación dejará de responder mientras la operación no haya terminado.

• **Rendimiento limitado**: En entornos de alta concurrencia (varios usuarios, múltiples peticiones de datos o eventos), el bloqueo puede provocar tiempos de respuesta muy elevados, afectando negativamente la experiencia del usuario y el rendimiento global de la aplicación.

¿Qué se entiende por operaciones asíncronas?

Las operaciones **asíncronas** permiten que el flujo de **ejecución no se bloquee** a la espera de que una tarea termine. En JavaScript, cuando nos encontramos con una operación asíncrona, la enviamos a un segundo plano (por ejemplo, el sistema operativo o una API), y seguimos ejecutando el resto del código sin esperar el resultado inmediato.

Ventajas de las operaciones asíncronas

• **No bloqueo**: El flujo de la aplicación no queda paralizado. JavaScript continúa ejecutando otras partes del código y, cuando la operación asíncrona concluye, el resultado se gestiona mediante funciones de devolución de llamada (callbacks), promesas o async/await.

• **Escalabilidad**: Permite manejar más peticiones y más usuarios al mismo tiempo, ya que las tareas no se “estancan” esperando la finalización de un proceso costoso.

Desventajas de las operaciones asíncronas

• **Complejidad de gestión**: Requiere una buena gestión de callbacks, promesas o uso de async/await para evitar anidar demasiadas operaciones y complicar la lectura (lo que en la jerga a veces se llama *callback hell*).

• **Dificultad de depuración**: Cuando ocurren errores en hilos asíncronos o en promesas, rastrear el punto de fallo exacto puede ser menos intuitivo que en un código puramente secuencial.

2.3. Ejemplos

Ejemplo 1: Operación síncrona de bloqueo

```
console.log("Inicio del programa");

// Simulamos una operación costosa con un bucle que tarde unos segundos
function operacionSincronaCostosa() {
  const start = Date.now();
  // Bucle de 2 segundos aproximadamente
  while (Date.now() - start < 2000) {
    // Simula una tarea muy pesada (cálculos intensivos)
  }
  console.log("Operación síncrona completada");
}
operacionSincronaCostosa();
console.log("Fin del programa");
```

1. El código escribe “Inicio del programa”.
2. Ejecuta la función *`operacionSincronaCostosa()`*, que **bloquea** el **hilo principal**, ya que el while impide que se ejecute cualquier otra instrucción hasta que se cumplan los 2 segundos de espera.
3. Tras finalizar la operación, se imprime *`“Operación síncrona completada”.`*
4. Finalmente, se escribe *`“Fin del programa”`*.

En este escenario, observamos claramente que el programa deja de responder durante esos 2 segundos, porque el **bucle** es **bloqueante** (hasta que no concluye, no se continúa con el flujo). Cualquier otra acción que intente realizar el usuario o el sistema tendrá que esperar.

Ejemplo 2: Operación asíncrona no bloqueante

```
console.log("Inicio del programa");


function operacionAsincronaCostosa(callback) {
   // Usamos setTimeout para simular una operación costosa que complete en 2s
  setTimeout(() => {
    console.log("Operación asíncrona completada");
    callback();
  }, 2000);
}
operacionAsincronaCostosa(() => {
  console.log("Callback ejecutado tras la operación asíncrona");
});
console.log("Fin del programa");
```

1. El código escribe *“Inicio del programa”*.
2. Llama a *operacionAsincronaCostosa()*, que internamente utiliza *setTimeout* para esperar 2 segundos. Sin embargo, el hilo principal **no se bloquea**.
3. Inmediatamente se imprime *“Fin del programa”*, reflejando que la ejecución continúa sin esperar los 2 segundos.
4. Pasados los 2 segundos, se ejecuta el código dentro de la función pasada como callback: primero se imprime *“Operación asíncrona completada”* y, a continuación, *“Callback ejecutado tras la operación asíncrona”*.

Con este comportamiento, cualquier otra parte de nuestro programa permanece libre para seguir ejecutándose. No hay un bloqueo del hilo principal, de modo que si surge alguna otra tarea o acción de usuario, el programa podría gestionarla mientras espera la finalización de la operación costosa.

2.4. Conclusiones

La distinción entre bloqueo y no bloqueo es esencial en **JavaScript**, sobre todo porque el lenguaje **se ejecuta en un solo hilo**, **denominado *event loop***. Cualquier operación que bloquee este único hilo de ejecución retrasará el resto de la aplicación, perjudicando su rendimiento y la experiencia del usuario. Por el contrario, las operaciones asíncronas aprovechan la naturaleza del *event loop* para delegar tareas costosas y permitir que el código principal fluya sin interrupciones.

En el desarrollo de aplicaciones web, especialmente en entornos donde se gestionan numerosas peticiones y eventos, ya sea en el lado cliente o en el lado servidor, **es muy importante identificar y tratar debidamente todas las operaciones susceptibles de bloqueo**. Adoptar un enfoque asíncrono garantiza escalabilidad, mayor capacidad de respuesta y una experiencia más fluida para el usuario final.

Para finalizar, debe quedar claro que JavaScript se inclina hacia el paradigma de la asíncronía y cuáles son las razones técnicas para optar por operaciones que no bloqueen la ejecución.

A lo largo de los próximos documentos, iremos profundizando en las diferentes técnicas y patrones de programación asíncrona (callbacks, promesas, async/await) y en cómo utilizarlos de forma eficiente y limpia en nuestros proyectos.

ASINCRONISMO EN JAVASCRIPT

TEMA 03 – EVENT LOOP

3.1. Introducción al Event Loop

JavaScript es un lenguaje de programación orientado a eventos y basado en un modelo de concurrencia peculiar, definido por el llamado ***Event Loop***. Conocer en profundidad cómo funciona el Event Loop es fundamental para entender el comportamiento de las aplicaciones JavaScript, especialmente cuando ejecutan operaciones asíncronas, gestionan múltiples eventos o manipulan elementos en el navegador sin bloquear la interfaz de usuario.

En este tema, exploraremos la mecánica interna del ***Event Loop***, su relación con la ***Call Stack (***pila de llamadas) y la ***Task Queue*** (cola de tareas), así como el concepto de ***microtareas*** y ***macrotareas***, piezas clave para comprender cómo y cuándo se ejecutan las operaciones en JavaScript.

3.2. Event Loop: Call Stack y Task Queue

¿Qué es el Event Loop?

El *Event Loop* (bucle de eventos) es el **mecanismo interno que utiliza JavaScript para coordinar la ejecución del código**, **gestionar las operaciones asíncronas** y orquestar la r**espuesta a eventos** en un único hilo de ejecución.

A diferencia de lenguajes que permiten la ejecución paralela en varios hilos, JavaScript se ejecuta en un solo hilo, por tanto, cuando se presentan **tareas** que deben gestionarse de manera **asíncrona** (como peticiones a un servidor o la espera de un temporizador), el **Event Loop** se encarga de **programarlas**, **atenderlas** en el momento adecuado **y regresar** el control al flujo principal.

Para entender cómo funciona, tenemos que entender como funcionan en conjunto los siguientes elementos:

Call Stack (pila de llamadas):

Es el lugar donde se almacenan las funciones que se están ejecutando en un momento dado. Cuando se llama a una función, esta se introduce en la Call Stack (push); al terminar su ejecución, se elimina de la misma (pop). En JavaScript, solo hay un hilo de ejecución, por lo que solo puede procesarse una instrucción a la vez.

API Web / APIs del entorno:

Aunque no es parte directa del Event Loop, las APIs del navegador (o del lado del servidor) cumplen la función de gestionar ciertas operaciones externas al lenguaje. Por ejemplo, si pides un setTimeout, la API de temporizador del navegador es la que maneja la cuenta atrás y, al cumplirse el intervalo, envía la tarea a la cola correspondiente para que el Event Loop la recoja y la ejecute cuando sea apropiado.

Task Queue (cola de tareas):

También conocida como *Message Queue* o *Callback Queue*, aquí se almacenan las *tareas* (eventos o funciones de callback) que están listas para ejecutarse una vez que la Call Stack se haya vaciado o quede libre. Cuando el Event Loop detecta que la Call Stack está vacía, toma la primera tarea de la Task Queue y la introduce en la Call Stack para su ejecución.

El proceso se repite de manera **cíclica**: siempre que la Call Stack está libre, el Event Loop consulta la Task Queue, coge la siguiente tarea pendiente y la ejecuta. Por eso lo llamamos un bucle de eventos (event loop).

¿Cómo interactúan Call Stack y Task Queue?

Cuando se ejecuta código JavaScript de manera **sincróna** (por ejemplo, simples operaciones matemáticas, manipulaciones de variables, llamadas directas a funciones), todo transcurre en la ***Call Stack*** de forma **lineal**: las funciones entran y salen de la pila sin mayor complejidad.

Sin embargo, cuando se invoca una función **asíncrona** (por ejemplo, realizar una petición fetch para obtener datos del servidor), esa función registra la operación en la API externa (la del navegador o la del servidor). Después de iniciada la operación, la ejecución en la Call Stack continúa con las siguientes líneas de código, sin esperar de manera bloqueante. Una vez que la operación **asíncrona** haya **finalizado** (por ejemplo, el servidor devuelve la respuesta), la tarea resultante (el callback con los datos recibidos) se envía a la Task Queue. Cuando el Event Loop verifica que la Call Stack está libre, introduce el callback en la Call Stack, permitiendo la ejecución de la función con los datos solicitados.

Este ciclo de *poner tareas en cola* y *sacarlas para ejecutarlas* es el corazón del modelo de concurrencia de JavaScript.

Aunque el lenguaje esté restringido a un solo hilo de ejecución, esta organización basada en el Event Loop le permite reaccionar ante múltiples eventos y manejar asíncronamente multitud de operaciones, siempre y cuando la Call Stack no esté ocupada procesando otras tareas.

3.3. Microtareas y Macrotareas

El modelo de concurrencia de JavaScript se enriquece con la distinción entre **dos tipos de tareas que se colocan en diferentes colas**: las ***microtareas*** y las ***macrotareas***. Esta división influye directamente en el orden en el que se ejecutan las callbacks.

¿Qué son las macrotareas?

Las *macrotareas* (o “tareas normales”) incluyen la mayoría de las operaciones que generamos en JavaScript mediante:

- Eventos del DOM (por ejemplo, click, keydown, etc.).
- Operaciones asíncronas con temporizadores: setTimeout, setInterval.
- Tareas de I/O (Entrada/Salida), como peticiones AJAX o fetch.

• Llamadas a APIs externas del navegador (por ejemplo, geolocalización).

Cada vez que se completa una macrotarea y la Call Stack queda libre, el Event Loop revisa si hay microtareas pendientes antes de tomar la siguiente macrotarea. De esta manera, **las macrotareas se van atendiendo en orden, pero siempre dejando espacio para que, en cada iteración del bucle, las microtareas se procesen primero**.

¿Que son las microtareas?

Las *microtareas* tienen prioridad sobre las macrotareas. Se ejecutan antes de que el Event Loop tome la siguiente macrotarea de la cola de tareas. Entre las operaciones que generan microtareas, encontramos:

• **Promesas resueltas**: cuando se cumple una promesa, se planifica su callback (.then, .catch) en la cola de microtareas.

• **Mutaciones del DOM**, como los producidos por la API MutationObserver, que permite detectar cambios en la estructura de un documento, como la inserción o eliminación de nodos.

• **Funciones asíncronas con async/await** cuando se resuelve la promesa interna.

El flujo de ejecución es el siguiente:

1. Ejecutar la macrotarea actual (por ejemplo, el resultado de un setTimeout).

2. Finalizar la macrotarea y vaciar todas las microtareas pendientes de la cola de microtareas antes de continuar.

3. Cuando no queden microtareas pendientes, el Event Loop pasa a la siguiente macrotarea de la cola de tareas.

Ejemplo ilustrativo con microtareas y macrotareas

Supongamos que tenemos este código simplificado:

```
console.log("Inicio");
// Programamos una macrotarea
setTimeout(() => {
  console.log("Macrotarea: setTimeout");
}, 0);
// Generamos una promesa que se resuelve de inmediato
Promise.resolve()
  .then(() => {
    console.log("Microtarea: Promesa resuelta");
  });
console.log("Fin");
```

1. **Se ejecuta de forma sincrónica**:

• *`console.log("Inicio")`* → Imprime *“Inicio”*.

• *`setTimeout(...)`* se programa en la API del navegador y, en cuanto cumple con el tiempo (0 ms en este caso), se agenda en la cola de macrotareas.

• *`Promise.resolve()`* se resuelve inmediatamente y se agenda su *`callback (.then(...))`* en la cola de microtareas.

• *`console.log("Fin")`* → *Imprime “Fin”*.

2. **Fin de la tarea actual (el script)**:

• Ahora que el script principal ha terminado, la Call Stack está libre.

3. **Event Loop atiende la cola de microtareas**:

• Dado que hay una microtarea pendiente (la promesa resuelta), esta se ejecuta primero. Así, se imprime “Microtarea: Promesa resuelta”.

4. **Event Loop atiende la cola de macrotareas**:

• Tras vaciar la cola de microtareas, se toma la primera macrotarea pendiente (la del setTimeout). Imprime “Macrotarea: setTimeout”.

La secuencia real de salida en consola es:

1. Inicio

2. Fin

3. Microtarea: Promesa resuelta

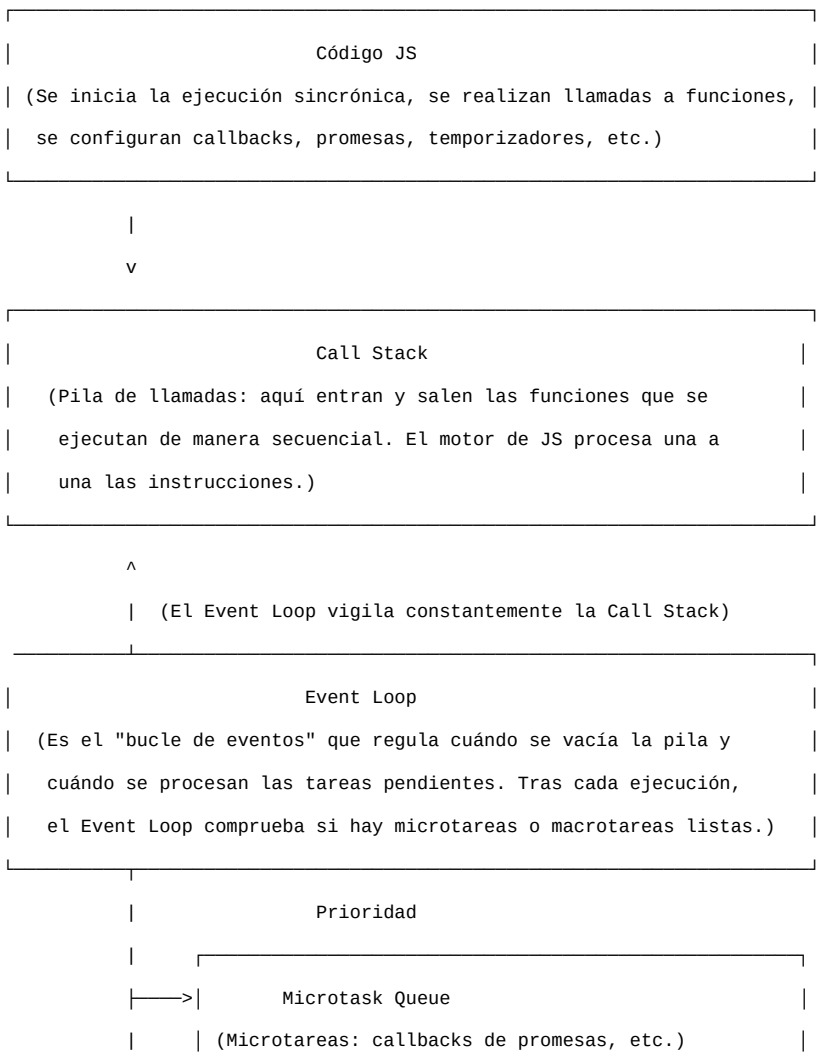
4. Macrotarea: setTimeout

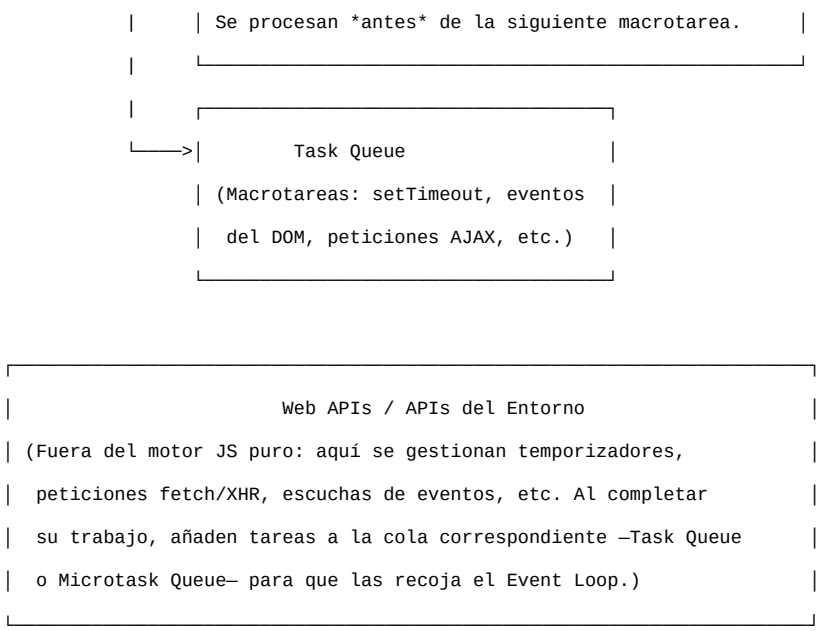
3.4. Representaciones gráficas del proceso

Esquema 1

A continuación, se presenta una representación gráfica que muestra de forma simplificada la relación entre la **Call Stack**, el **Event Loop**, la **Task Queue** y los dos tipos de tareas (**macrotareas** y **microtareas**). Asimismo, se ilustra el papel de las **Web APIs** (o APIs del entorno), que actúan fuera del motor de JavaScript pero cooperan para manejar las operaciones asíncronas.

Cada bloque representa un “espacio” donde ocurren procesos concretos. Las flechas indican el flujo de tareas:





Código JS:

Aquí se lee y ejecuta el script principal de forma síncrona. Si hay llamadas a funciones, promesas, setTimeout, etc., se registran en las **Web APIs** o se añaden posteriormente a las colas correspondientes.

Call Stack (Pila de Llamadas):

Todas las funciones que se están ejecutando en un momento dado van pasando por la pila. Cuando invocas una función, esta se apila y, al terminar, se desapila. JavaScript es monohilo , por lo que únicamente puede procesar una cosa a la vez en esta pila.

Event Loop (Bucle de Eventos):

Es un proceso que constantemente revisa si la **Call Stack** está vacía. Si lo está, el Event Loop comprueba primero si existen **microtareas** pendientes en la **Microtask Queue**. De haberlas, las ejecuta.

Después de procesar todas las microtareas, toma la siguiente **macrotarea** de la **Task Queue** y la coloca en la **Call Stack**.

Microtask Queue:

Aquí se encolan las microtareas, que tienen prioridad sobre las macrotareas. Principalmente se generan al resolverse promesas (Promise.resolve(), métodos .then(), async/await) y ciertos cambios internos del DOM.

Antes de procesar la siguiente macrotarea, JavaScript vacía por completo la cola de microtareas.

Task Queue (cola de Macrotareas):

Incluye las tareas relacionadas con setTimeout, setInterval, peticiones AJAX o fetch, e incluso eventos del DOM (click, submit, etc.). Tras cada ciclo del Event Loop, una vez procesadas las microtareas, se atiende la siguiente macrotarea.

Web APIs / **APIs del entorno**:

No forman parte directamente del motor JavaScript, sino que son utilidades ofrecidas por el entorno (un navegador o Node.js). Por ejemplo, cuando llamas a setTimeout, la cuenta atrás la gestiona el navegador. Una vez cumplido el tiempo, la API añade la callback a la **Task Queue** (cola de macrotareas) para que el Event Loop la ejecute cuando la Call Stack esté libre.

Resumen de ideas clave

• La *Call Stack* ejecuta las funciones una detrás de otra.

• Las *Web APIs* (en un navegador) y las APIs del entorno (en Node.js) gestionan las operaciones asíncronas.

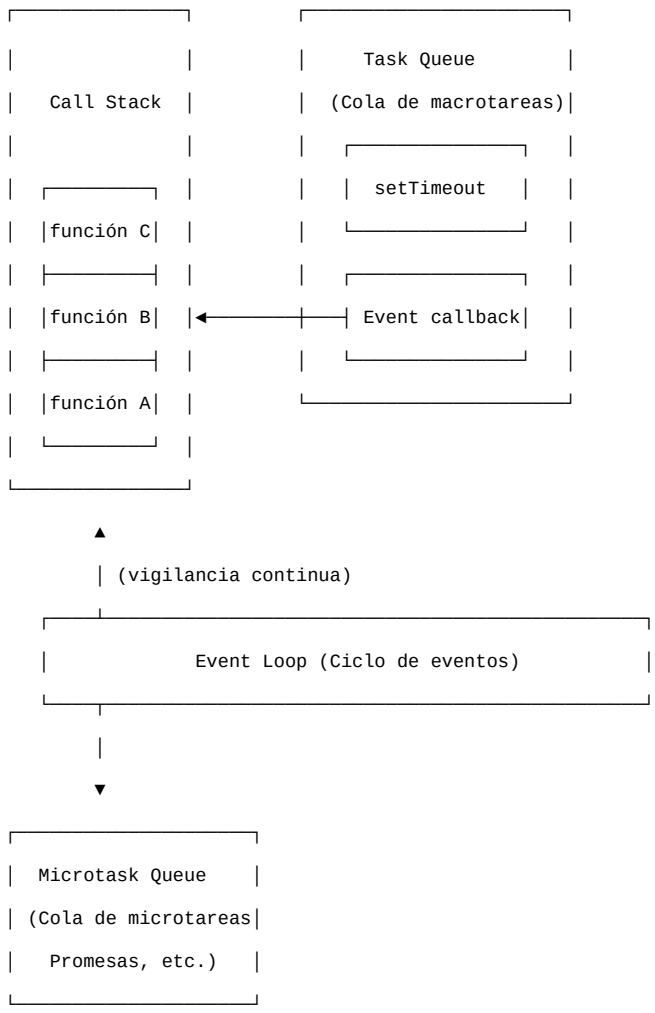
• Cuando acaban, dichas operaciones colocan sus callbacks en la Task Queue o la Microtask Queue.

• El *Event Loop* controla el orden: primero ejecuta todo lo que haya en la Microtask Queue (si hay microtareas pendientes), y luego pasa a tomar la siguiente macrotarea en la Task Queue.

• Este mecanismo permite que JavaScript maneje de forma asíncrona múltiples eventos y tareas sin bloquear el hilo principal, pese a ser un lenguaje monohilo.

Esquema 2

Otra forma de ver estos procesos de interacción es la que se muestra a continuación



Call Stack (Pila de llamadas)

La **Call Stack** es el corazón de la ejecución secuencial en JavaScript. En ella se apilan y desapilan las funciones conforme el flujo de ejecución avanza.

En el diagrama, la Call Stack se representa como una columna en la que pueden encontrarse varias funciones (A, B, C, etc.) según vayan

siendo llamadas. Solo podemos procesar una tarea a la vez dentro de la Call Stack.

Task Queue (Cola de macrotareas)

La **Task Queue** (también conocida como *Message Queue*) es donde se encolan las *macrotareas*.

En el diagrama, dentro de la **Task Queue**, se observan ejemplos de tareas como setTimeout y Event callback. Cuando dichas tareas finalizan su gestión en las Web APIs (por ejemplo, el temporizador cuenta el tiempo y, al llegar a cero, empuja la callback a la cola), se colocan en la Task Queue.

Microtask Queue (Cola de microtareas)

La **Microtask Queue** es donde se almacenan las *microtareas*, que suelen tener prioridad sobre las macrotareas.

Entre las principales fuentes de microtareas destacan las promesas resueltas: cuando llamamos a `Promise.resolve()` o en cuanto se cumple la promesa en una llamada `fetch` con `.then()`, el callback se agenda en la cola de microtareas. También las operaciones internas de mutación del DOM, en ciertos contextos, se tratan como microtareas.

En el diagrama, este bloque se dibuja en la parte inferior, indicando que, cuando el Event Loop detecta microtareas pendientes, las ejecuta *antes* de pasar a la siguiente macrotarea.

Event Loop (Ciclo de eventos)

El **Event Loop** es un proceso que supervisa de manera continua el estado de la Call Stack y de las colas de tareas (principalmente, la Task Queue y la Microtask Queue).

Su dinámica de trabajo es la siguiente:

1. **Observa la Call Stack**. Si la Call Stack está vacía o acaba de terminar de procesar la función en curso, el Event Loop revisa qué tareas están pendientes.

2. **Revisa primero la Microtask Queue**: si hay microtareas, las ejecuta todas en orden hasta que la cola de microtareas quede vacía.

3. **Pasa a la Task Queue**: si ya no hay microtareas pendientes, toma la primera macrotarea de la Task Queue (si existe) y la introduce en la Call Stack para su ejecución.

4. **Se repite el ciclo**: una vez procesada la macrotarea, vuelve a revisar la Microtask Queue y repite el procedimiento.

Dicho de otra forma, **cada vez que la Call Stack se vacía**, se verifica primero si hay microtareas en la Microtask Queue (ya que tienen prioridad). Una vez gestionadas, se atiende la siguiente macrotarea en la Task Queue.

Interacción con las APIs del navegador (o APIs de Node.js)

Aunque no aparece explícito en el diagrama, es importante señalar que buena parte del trabajo asíncrono se inicia a través de las APIs del entorno (por ejemplo, en un navegador, la API de setTimeout, fetch, eventos del DOM, etc.).

Al completarse la operación asíncrona (por ejemplo, si se cumplen los milisegundos en setTimeout o llega la respuesta de un servidor con fetch), las APIs externas añaden el callback correspondiente a la cola apropiada (Task Queue o Microtask Queue).

3.5. Ejemplos

Ejemplo 1: Call Stack

La **Call Stack** registra en qué orden se invocan las funciones y cómo estas van entrando y saliendo de la pila. En un entorno monohilo como JavaScript, solo podemos ejecutar una función a la vez.

```
// Ejemplo 1: Call Stack básico
function funcionC() {
  console.log("funcionC: Ha sido llamada y está en la cima de la Call Stack");
}
function funcionB() {
  console.log("funcionB: Se llama a funcionC");
  funcionC();
  console.log("funcionB: Ha regresado de funcionC");
}
function funcionA() {
  console.log("funcionA: Se llama a funcionB");
  funcionB();
  console.log("funcionA: Ha regresado de funcionB");
}
console.log("Inicio de la ejecución principal ...");
funcionA();
console.log("Fin de la ejecución principal");
```

Explicación

1. El flujo comienza por la línea *`console.log("Inicio de la ejecución principal...")`*.

2. Llamamos a *`funcionA()`*, que se **apila** en la Call Stack.

3. Dentro de *`funcionA()`*, se llama a *`funcionB()`*, que entra en la Call Stack por encima de *`funcionA()`*.

4. Dentro de *`funcionB()`*, se llama a *`funcionC()`*, que se apila sobre todas las anteriores.

5. Cuando *`funcionC()`* termina, se desapila y regresa el control a *`funcionB()`*, que posteriormente se desapila y regresa el control a *`funcionA()`*.

6. Finalmente, *`funcionA()`* se desapila y volvemos al ámbito global, donde ejecutamos el último *`console.log`*.

Ejemplo 2: Macrotareas

Las macrotareas se generan cuando utilizamos, por ejemplo, *`setTimeout`*, *`setInterval`* o eventos del DOM. Una vez la operación asíncrona está lista, su callback se encola en la Task Queue, esperando a que el Event Loop la pase a la Call Stack cuando esta esté libre.

```
// Ejemplo 2: Macrotarea con setTimeout
console.log("1) Inicio del script");
setTimeout(() => {
      console.log("3) Esta es la macrotarea (setTimeout) ejecutándose");
}, 0);
console.log("2) Fin del script (código sincrónico)");
```

Explicación

1. Se ejecuta primero todo el código sincrónico: imprime “**1) Inicio del script”** y luego “**2) Fin del script”**.

2. setTimeout programa una macrotarea con retardo 0. Sin embargo, **esto no significa ejecución inmediata**, sino que se coloca en la cola de macrotareas.

3. Cuando la Call Stack está libre (tras el paso 2), el Event Loop recoge la macrotarea de setTimeout y ejecuta el callback, que imprime “**3) Esta es la macrotarea (setTimeout) ejecutándose”**.

El resultado en consola es:

```
1) Inicio del script
2) Fin del script
3) Esta es la macrotarea (setTimeout) ejecutándose
```

Ejemplo 3: Microtareas

Las microtareas suelen provenir de promesas resueltas. Tienen prioridad sobre las macrotareas: una vez se vacía la Call Stack, se procesan todas las microtareas pendientes antes de atender la siguiente macrotarea.

```
// Ejemplo 3: Microtareas con Promesas
console.log("A) Inicio sincrónico");
Promise.resolve()
  .then(() => {
    console.log("B) Promesa resuelta (microtarea 1)");
  })
  .then(() => {
        console.log("C) Encadenada tras la promesa anterior (microtarea 2)");
  });
console.log("D) Fin sincrónico");
```

Explicación

1. Se imprimen en primer lugar “**A) Inicio sincrónico”** y “**D) Fin sincrónico”**, porque las promesas se resuelven de manera asíncrona.

2. Una vez el script principal termina (la Call Stack se vacía), el Event Loop atiende la **cola de microtareas**. La promesa ya está resuelta, así que se ejecuta la primera *`callback (console.log("B) Promesa resuelta..."))`*.

3. Inmediatamente después, se encadena la siguiente microtarea (.then(...)), que imprime “**C) Encadenada tras la promesa anterior…”**.

El orden de salida en la consola es:

```
A) Inicio sincrónico
D) Fin sincrónico
B) Promesa resuelta (microtarea 1)
C) Encadenada tras la promesa anterior (microtarea 2)
```

3.6. Conclusión y relevancia en la práctica

Comprender esta arquitectura interna —monohilo con un *Event Loop* que coordina colas de macrotareas y microtareas— ayuda a explicar por qué JavaScript no se bloquea al esperar respuestas de un servidor o por un temporizador: esas operaciones se “desplazan” fuera del motor principal y se reinsertan como tareas cuando están listas.

Asimismo, es crucial para entender el orden de ejecución de callbacks, especialmente con promesas y async/await, y para evitar comportamientos inesperados (por ejemplo, por qué cierto console.log aparece antes que otro en la consola, a pesar de que su línea de código está aparentemente después).

Finalmente, los diagramas y ejemplos mostrados evidencian la relación entre la **Call Stack**, la **Task Queue**, la **Microtask Queue** y el **Event Loop**, pilares fundamentales de la programación asíncrona en JavaScript. Cuando se dominan estos conceptos, resulta más sencillo identificar y predecir el flujo de ejecución del código, optimizar la respuesta de una aplicación y solucionar problemas o cuellos de botella que puedan surgir en el desarrollo de proyectos web.

ASINCRONISMO EN JAVASCRIPT

TEMA 04 – CALLBACKS

4.1. Introducción a los callbacks

En JavaScript, gran parte de la potencia y flexibilidad del lenguaje radica en su capacidad para trabajar de forma asíncrona, es decir, para llevar a cabo operaciones en segundo plano sin bloquear la ejecución principal de un programa. Una de las primeras y más tradicionales formas de gestionar estas operaciones asíncronas en JavaScript es mediante el uso de *callbacks*.

A lo largo de esta sección, abordaremos en profundidad qué son los callbacks, cómo funcionan en el lenguaje y cuáles son los inconvenientes que pueden surgir al utilizarlos de manera excesivamente anidada, fenómeno conocido como *Callback Hell*.

4.2. ¿Qué son los callbacks y cómo funcionan?

Un *callback* es, en esencia, una función que se pasa como argumento a otra función para que esta la ejecute más tarde, ya sea después de completar una tarea asíncrona o en respuesta a un evento específico. Dicho de otra manera, un callback sirve para indicarle a una función qué hacer una vez que haya terminado una cierta operación (por ejemplo, una llamada a un servidor, la lectura de un archivo, o un temporizador).

En JavaScript, las funciones se pueden:

1. **Asignar** a una variable.
2. **Pasar** como argumento a otra función.
3. **Devolver** desde una función.

Esta característica del lenguaje facilita la creación y el uso de funciones de callback. El patrón típico de uso es:

1. Se define una función principal que realiza una operación (por ejemplo, solicitar datos a un servidor).

2. Se pasa como argumento una segunda función (callback) para que la primera la invoque una vez que termine su tarea.

Veamos un ejemplo sencillo que muestra cómo funcionan las callbacks:

```
// Función que simula una tarea asíncrona, por ejemplo, consulta a una API
function getDataFromServer(callback) {
  console.log("Iniciando la petición al servidor...");
  // Usamos un setTimeout para simular un retardo
  setTimeout(function() {
    const data = { nombre: "Juan", edad: 30 };
    console.log("Datos obtenidos del servidor.");
    // Llamamos a la función de callback con los datos
    callback(data);
  }, 2000);
}
// Función que se utilizará como callback para manejar los datos
function processData(data) {
  console.log("Procesando los datos recibidos...");
  console.log(`Nombre: ${data.nombre}, Edad: ${data.edad}`);
}
// Invocamos la función principal y le pasamos 'processData' como callback
getDataFromServer(processData);
```

En este ejemplo:

• getDataFromServer(callback) simula una operación asíncrona que, en la vida real, podría ser una petición a un servidor externo.

• Dentro de setTimeout, pasado cierto tiempo, se obtienen supuestamente unos datos.

• Una vez disponibles los datos, la función callback (en este caso processData) se invoca y recibe esos datos como argumento.

• processData a su vez muestra la información por consola.

Orden de ejecución

Un detalle crucial para entender cómo funcionan los callbacks es el orden en el que se ejecutan. JavaScript, a pesar de ser un lenguaje monohilo, cuenta con un *Event Loop* que gestiona las operaciones asíncronas. Cuando setTimeout finaliza el tiempo de espera, la función de callback que se le ha pasado se coloca en la cola de tareas (Task Queue).

En cuanto el *Event Loop* detecta que la pila de ejecución está vacía, transfiere esa función de la cola a la pila para que finalmente se ejecute. Este proceso explica por qué la parte sincrónica del código (instrucciones que no dependen de la operación asíncrona) se ejecuta primero, mientras que la ejecución de la función callback se realiza después.

Uso de callbacks en eventos

Otra faceta muy habitual de los callbacks en JavaScript se da en la interacción con el DOM (Documento de Modelo de Objetos) y la gestión de eventos. Por ejemplo, se puede asignar una función callback que se dispare al hacer clic en un botón:

```
<button id="miBoton">Haz clic aquí</button>
<script>
  const boton = document.getElementById("miBoton");
  // Definimos la función callback que se ejecutará tras el clic
  function manejarClick() {
    alert("¡Has hecho clic en el botón!");
  }
  // Añadimos la función callback como listener del evento 'click'
  boton.addEventListener("click", manejarClick);
</script>
```

Este tipo de patrones es muy común en la programación web, donde se delega a las callbacks la responsabilidad de reaccionar a eventos externos como clics de ratón, pulsaciones de teclado, envío de formularios, etc.

4.3. Problemas del “Callback Hell”

La versatilidad de los callbacks hizo que durante mucho tiempo fueran la principal forma de controlar la asincronía en JavaScript. Sin embargo, a medida que las aplicaciones crecieron en complejidad, se popularizó un término para describir la excesiva anidación de funciones callback: el llamado *Callback Hell*.

¿Qué es el Callback Hell?

El *Callback Hell* ocurre cuando, para llevar a cabo una secuencia de operaciones asíncronas, vamos encadenando múltiples callbacks de manera anidada. El resultado es un código que se ve cada vez más inclinado hacia la derecha, difícil de leer y, sobre todo, muy complicado de mantener o depurar. Un ejemplo muy caricaturesco de *Callback Hell* puede lucir así:

```
funcionAsincrona1(function(resultado1) {
  funcionAsincrona2(resultado1, function(resultado2) {
    funcionAsincrona3(resultado2, function(resultado3) {
      funcionAsincrona4(resultado3, function(resultado4) {
        // ...
        console.log("Demasiado anidado y difícil de seguir...");
      });
    });
  });
});
```

Aunque este ejemplo es intencionadamente exagerado, es frecuente encontrarse con estructuras similares en proyectos grandes si no se siguen algunas pautas de organización del código.

Por qué es problemático

Legibilidad reducida:

A medida que se añaden más callbacks, el código se va moviendo hacia la derecha y su estructura se vuelve más compleja y menos intuitiva. Esto dificulta entender de un vistazo qué está ocurriendo y en qué orden.

Mantenimiento complicado:

Con muchos niveles de anidación, cualquier cambio en la lógica (por ejemplo, añadir una nueva condición o una verificación de errores en algún punto intermedio) puede requerir reestructurar por completo el bloque de callbacks.

Gestión de errores confusa:

En ocasiones, la información de error debe pasar a través de múltiples funciones para tratar cada situación. Controlar de manera ordenada cada posible fallo puede volverse muy engorroso cuando cada nivel de callback debe gestionar excepciones o devolver información de error.

Buenas prácticas para evitar el Callback Hell

Aunque la sección se enfoca en los callbacks, es relevante mencionar algunas pautas que tradicionalmente se han usado para mantener el código limpio y evitar caer en el *Callback Hell*:

Modularizar las funciones:

En lugar de definir la lógica completa dentro de un único callback, se puede dividir en funciones más pequeñas, cada una encargada de una parte de la operación, y luego encadenarlas con nombres descriptivos.

Manejar errores de forma centralizada:

Definir una convención clara para manejar errores en los callbacks. Por ejemplo, la convención node.js de callback(error, data), donde el primer parámetro es el posible error y el segundo son los datos de la operación, ayuda a estandarizar la gestión de fallos.

Usar Promesas y posteriormente async/await:

Mediante el uso de promesas y el de async/await, el código asíncrono se ha vuelto más lineal y más fácil de manejar. Aunque estos temas se abordan en secciones dedicadas, conviene destacar que, en la práctica

actual, se recomienda migrar o arrancar los proyectos directamente con Promesas o con async/await, dejando los callbacks para casos puntuales.

En conclusión, los callbacks constituyen la base histórica del manejo de asincronía en JavaScript. Son una herramienta potente, pero el abuso o la ausencia de patrones claros al emplearlos puede derivar en *Callback Hell*, dificultando la legibilidad y el mantenimiento de nuestro código. Por ello, es fundamental conocer sus ventajas y desventajas, y sobre todo, estar familiarizados con las mejoras y evoluciones que ha tenido el lenguaje para manejar la asincronía de manera más elegante y limpia (como Promesas o async/await). Conociendo estos conceptos, podremos crear aplicaciones JavaScript más robustas y fáciles de mantener.

4.4. Ejemplos

Ejemplo 1: callback sincrónico

Aunque la esencia de los callbacks en JavaScript está muy ligada a la asincronía, es posible utilizar esta técnica de manera “sincrónica” para fines didácticos. En este ejemplo, simplemente pasamos una función como parámetro a otra función y la ejecutamos inmediatamente:

```
/** Función que recibe un callback y lo ejecuta inmediatamente
 * con un mensaje. */
function ejecutaCallback(callback) {
  const mensaje = "Saludo desde ejecutaCallback";
  callback(mensaje);
}
// Definimos el callback que recibirá el mensaje
function mostrarMensaje(texto) {
  console.log("Callback ejecutado: " + texto);
}
// Llamamos a la función principal pasando 'mostrarMensaje' como
callback
ejecutaCallback(mostrarMensaje);
```

Explicación:

1. La función ejecutaCallback genera un mensaje en una variable local.

2. Invoca el callback mostrarMensaje pasándole ese mensaje.

3. El callback se ejecuta y muestra el texto por consola.

Este ejemplo es puramente ilustrativo y se centra en mostrar el paso de funciones como argumentos, sin la parte asíncrona tan típica de JavaScript.

Ejemplo 2: callback asíncrono

En este caso, aprovechamos la función setTimeout para simular una llamada asíncrona. Una vez transcurrido el tiempo especificado, se invoca la función de callback:

```
/** Función que simula un proceso asíncrono con setTimeout.
 * Recibe un callback que se ejecutará después de 2 segundos. */
function procesoAsincrono(callback) {
  console.log("Proceso iniciado...");
  setTimeout(function() {
    const resultado = "Datos obtenidos tras un tiempo de espera";
    // Invocamos el callback pasándole el resultado
    callback(resultado);
  }, 2000);
}
// Definimos el callback para manejar el resultado
function manejarResultado(dato) {
  console.log("Callback manejando el resultado: " + dato);
}
// Llamamos a la función principal y pasamos nuestro callback
procesoAsincrono(manejarResultado);
```

Explicación

1. console.log("Proceso iniciado...") se ejecuta primero.

2. Después de 2 segundos, manejarResultado se invoca con el mensaje "Datos obtenidos tras un tiempo de espera".

3. Este patrón refleja la base del manejo asíncrono con callbacks en JavaScript.

Ejemplo 3: callback hell

El *Callback Hell* se produce cuando tenemos una serie de operaciones asíncronas que dependen unas de otras y las encadenamos de manera excesivamente anidada. A continuación, se muestra un ejemplo intencionadamente exagerado para ilustrar cómo se ve un código difícil de mantener:

```
// Simulamos funciones asíncronas anidadas
function operacion1(datos, callback) {
  setTimeout(function() {
    console.log("Operación 1 completada con:", datos);
    callback(datos + " -> O1");
  }, 1000);
}
function operacion2(datos, callback) {
  setTimeout(function() {
    console.log("Operación 2 completada con:", datos);
    callback(datos + " -> O2");
  }, 1000);
}
function operacion3(datos, callback) {
  setTimeout(function() {
    console.log("Operación 3 completada con:", datos);
    callback(datos + " -> O3");
  }, 1000);
}

// Aquí vemos la anidación que provoca el 'Callback Hell'
operacion1("Iniciando", function(resultado1) {
  operacion2(resultado1, function(resultado2) {
```

```
    operacion3(resultado2, function(resultado3) {
      console.log("Resultado final tras múltiples operaciones: " +
resultado3);
      // Podríamos seguir anidando más y más...
    });
  });
});
```

Problemas que se encuentran:

Excesiva indentación: El código se desplaza gradualmente hacia la derecha, dificultando la legibilidad.

Dificultad de mantenimiento: Si quisiéramos modificar alguna parte del flujo o controlar errores en cada paso, se complicaría notablemente.

Escalabilidad limitada: Añadir más pasos requiere anidar más callbacks, empeorando la situación.

Ahora vamos a ver dos ejemplos más completos donde combinamos tanto la parte teórica de los *callbacks* como la posibilidad de caer en un escenario de *Callback Hell*. El objetivo es mostrar cómo interactúan varias funciones asíncronas que comparten o transforman datos.

Ejemplo 4: Envío de datos a un servidor y notificación al usuario

Supongamos que tenemos estas tareas:

1. Recoger datos de un formulario.
2. Enviarlos al servidor (simulado con setTimeout).
3. Mostrar una notificación tras el envío.

```
/** Simula la recolección de datos de un formulario. Como podría
ser rápido o lento, se gestiona asíncronamente con setTimeout. */
function recogerDatosFormulario(callback) {
  console.log("Recogiendo datos del formulario...");
```

```
/** Suponemos un retardo de 1s para simular que puede tardar en completarse */
  setTimeout(function() {
    const datos = {
      nombre: "Laura",
      email: "laura@example.com"
    };
    console.log("Datos del formulario obtenidos.");
    callback(datos);
  }, 1000);
}
/** Simula el envío de datos a un servidor. Cuando finaliza, invoca el callback. */
function enviarDatosAServidor(datos, callback) {
     console.log(`Enviando los datos de ${datos.nombre} al servidor...`);
  // Simulamos un retardo de 2s
  setTimeout(function() {
             console.log(`Datos de ${datos.nombre} enviados correctamente.`);
    callback();
  }, 2000);
}
// Simula la notificación al usuario tras completar el envío.
function notificarUsuario() {
     console.log("Notificación: El proceso ha finalizado con éxito.");
}
// Encadenamos las llamadas utilizando callbacks
recogerDatosFormulario(function(datosRecibidos) {
  enviarDatosAServidor(datosRecibidos, function() {
    notificarUsuario();
  });
});
```

Explicación:

1. recogerDatosFormulario recoge (de manera simulada) los datos e invoca el callback con dichos datos.

2. En la función anónima que recibe datosRecibidos, se llama a enviarDatosAServidor, pasándole datosRecibidos y otra función de callback.

3. Tras completar el envío, se llama a notificarUsuario.

Este flujo no es muy extenso, pero si añadimos más pasos asíncronos (validación, transformaciones, consultas adicionales, etc.), se incrementaría la probabilidad de caer en un *Callback Hell*.

Ejemplo 5: Cadena de validaciones con anidación

En este segundo ejemplo, mostraremos cómo varias validaciones consecutivas pueden incrementar la complejidad del código si seguimos usando solo callbacks sin una estructura clara.

```
// Validación 1: Comprobar que el usuario no está bloqueado
function checkUserNotBlocked(usuario, callback) {
  setTimeout(function() {
    if (usuario.bloqueado) {
      return callback("Usuario bloqueado, no puede continuar.");
    }
    console.log("Validación 1: El usuario no está bloqueado.");
    callback(null, usuario);
  }, 1000);
}
// Validación 2: Comprobar que el usuario tiene permisos
function checkUserPermissions(usuario, callback) {
  setTimeout(function() {
    if (!usuario.permisos.includes("ADMIN")) {
```

```
      return callback("El usuario no tiene permisos suficientes.");
    }
    console.log("Validación 2: El usuario posee los permisos necesarios.");
    callback(null, usuario);
  }, 1000);
}
// Validación 3: Comprobar que la suscripción está activa
function checkSubscription(usuario, callback) {
  setTimeout(function() {
    if (!usuario.suscripcionActiva) {
      return callback("La suscripción del usuario no está activa.");
    }
    console.log("Validación 3: La suscripción del usuario está activa.");
    callback(null, usuario);
  }, 1000);
}
// Simulamos un objeto 'usuario'
const usuarioEjemplo = {
  nombre: "Carlos",
  bloqueado: false,
  permisos: ["ADMIN", "EDITOR"],
  suscripcionActiva: true
};
/** Llamadas encadenadas: Ejemplo que puede complicarse conforme crezcan las validaciones */
checkUserNotBlocked(usuarioEjemplo, function(error, usuario1) {
  if (error) {
    return console.error("Error:", error);
  }
  checkUserPermissions(usuario1, function(error, usuario2) {
    if (error) {
      return console.error("Error:", error);
```

```
  }
  checkSubscription(usuario2, function(error, usuario3) {
    if (error) {
      return console.error("Error:", error);
    }
    console.log(
      "Todas las validaciones pasadas con éxito para el usuario:",
      usuario3.nombre
    );
  });
 });
});
```

Explicación:

1. Cada validación simula un proceso asíncrono (con setTimeout).

2. Si hay un error, se retorna inmediatamente para interrumpir el flujo.

3. En caso de éxito, se pasa el usuario al siguiente callback hasta completar todas las validaciones.

4. A medida que se van añadiendo más verificaciones, el anidamiento crece y complica la lectura, ilustrando la problemática típica del *Callback Hell*.

4.5. Conclusión

Estos ejemplos reflejan las bondades y los desafíos de los *callbacks*. Han sido la base para manejar la asincronía en JavaScript durante mucho tiempo, pero, como hemos visto, pueden derivar en un *Callback Hell* que resulte poco mantenible. Por eso, en la actualidad se recomienda hacer uso de promesas y, especialmente, de la sintaxis async/await para escribir código más legible, aunque conocer los callbacks sigue siendo esencial para comprender el modelo de ejecución de JavaScript y mantener proyectos que aún los utilicen.

Es fundamental el conocimiento de estos patrones de programación asíncrona, tanto en su versión con callbacks como en la posterior evolución

con Promesas y async/await, con el fin de adquirir una comprensión completa del asincronismo en JavaScript.

ASINCRONISMO EN JAVASCRIPT

TEMA 05 – APIs NATIVAS PARA ASINCRONISMO

5.1. Introducción

Existen varias formas de trabajar con asincronismo en JavaScript:

• **Callbacks** (la forma más antigua y básica).

• **Web APIs** y librerías del navegador que nos permiten programar comportamientos asíncronos, principalmente:

- setTimeout y setInterval.
- fetch.
- Xmlhttprequest.
- Promises.
- async/await.

En este documento, nos centraremos en describir setTimeout y setInterval como ejemplos de temporizadores de JavaScript, así como en introducir la función fetch para la realización de solicitudes HTTP (peticiones a un servidor remoto para obtener o enviar datos)

5.1. Uso de setTimeout y setInterval

¿Qué son?

• setTimeout: Programa la ejecución de un código (función o fragmento de código) transcurrido un período de tiempo que se especifica en milisegundos. Es decir, se utiliza para retrasar la ejecución de algo.

• setInterval: Programa la ejecución periódica de un código, de modo que se ejecuta una vez transcurrido el período de tiempo especificado y se repite de forma indefinida (o hasta que lo detengamos).

La clave para entender por qué se consideran APIs asíncronas es que, cuando se programa una tarea con setTimeout o setInterval, JavaScript no se queda “esperando” a que pase el tiempo indicado, sino que registra el evento en la cola del **event loop**. Mientras, el intérprete sigue ejecutando el resto del script. Una vez que se cumpla el plazo en milisegundos establecido, el motor de JavaScript recuperará la función o código asociado y lo ejecutará.

Cómo se relacionan setTimeout y setInterval con el event loop y la task queue

setTimeout

Cuando usas setTimeout(callback, tiempo):

1. Se registra un temporizador y se indica que, **una vez transcurridos** los milisegundos especificados (tiempo), se añada la función callback a la cola de tareas.

2. **Importante**: el temporizador no “garantiza” que el callback se ejecute exactamente al milisegundo exacto. Solo se asegura de que, como mínimo, pasen esos milisegundos antes de que el callback ingrese a la cola de tareas.

3. Pasado el tiempo configurado, se añade la tarea (el callback) a la **task queue**.

4. El **event loop**, cuando termine de ejecutar la pila principal, atenderá esta nueva tarea. Si en ese momento no hay nada de mayor prioridad, tomará la tarea de la cola y la ejecutará.

Ejemplo

```
console.log("Inicio");
setTimeout(() => {
  console.log("Mensaje retrasado");
}, 2000);
console.log("Fin");
```

• console.log("Inicio") y console.log("Fin") se ejecutan de inmediato (sincrónicamente).

• El setTimeout se configura para 2 segundos. Cuando estos transcurran, el callback (() => { console.log("Mensaje retrasado") }) se envía a la **task queue**.

• El event loop lo ejecutará en cuanto finalicen todas las tareas sincrónicas que se encuentren en la cola o en el stack.

setInterval

setInterval(callback, tiempo) funciona de forma similar a setTimeout, pero con la diferencia de que, al cumplirse el intervalo de tiempo, **la tarea (callback) se vuelve a programar** para ser ejecutada de nuevo al cabo de otros tantos milisegundos. Esto se repite indefinidamente o hasta que sea cancelado con clearInterval. El proceso es:

1. Se configura un intervalo repetitivo cada tiempo milisegundos.

2. Cada vez que transcurre ese lapso, el **motor de JavaScript** añade el callback a la cola de tareas.

3. El **event loop** extrae el callback de la cola y lo ejecuta cuando el call stack está libre.

Ejemplo

```
let contador = 0;
const idIntervalo = setInterval(() => {
  contador++;
  console.log(`Contador: ${contador}`);
  if (contador >= 5) {
    clearInterval(idIntervalo);
    console.log("Intervalo detenido.");
  }
}, 1000);
```

• Cada 1000 milisegundos, la tarea se añade a la cola de tareas.

• El event loop la atiende cuando puede (normalmente de inmediato si no hay otra tarea anterior).

• Se incrementa el contador y se muestra en consola.

• Al llegar a 5, se cancela el intervalo.

Diferencias y consideraciones

1. **No bloquean el hilo**: Ni setTimeout ni setInterval bloquean la ejecución del resto de código. Crean “tareas pendientes” que se ponen en la cola de tareas. El motor de JavaScript puede seguir ejecutando otra lógica mientras espera a que se cumpla el tiempo o el intervalo.

2. **Orden de ejecución**: Las tareas asíncronas solo se ejecutan cuando el **call stack está vacío**. Esto implica que, si el programa principal está haciendo un cálculo muy pesado, puede haber un retraso adicional.

3. **Precisión**: El tiempo de retraso es orientativo. No está garantizado que se cumpla exactamente, ya que depende de la carga que el hilo principal tenga en ese momento, aunque de forma habitual, la precisión es razonable.

4. **Identificador de intervalo**: Tanto setInterval como setTimeout devuelven un identificador (un número entero) que sirve para detener la ejecución programada. Para setTimeout, se usa clearTimeout(), y para setInterval, se usa clearInterval().

Tanto setTimeout como setInterval programan ejecuciones asíncronas controladas por el event loop, utilizando la task queue como lugar de espera antes de su ejecución. Esta arquitectura permite que JavaScript siga ejecutando el resto del programa mientras esperan el tiempo establecido o mientras se planifican repeticiones periódicas, ofreciendo una experiencia fluida y no bloqueante.

5.2. API fetch para solicitudes HTTP

¿Qué es fetch?

La API fetch es una de las Web APIs más comunes para realizar solicitudes HTTP desde JavaScript. En versiones antiguas de JavaScript, se usaba XMLHttpRequest (XHR) para realizar peticiones AJAX, pero fetch ha ido ganando terreno gracias a su diseño basado en Promesas, que facilita la lectura y escritura del código.

Sintaxis general:

```
fetch(url, [opciones])
  .then(respuesta => {
    // Procesar la respuesta
  })
  .catch(error => {
    // Manejar el error
  });
```

Donde:

• url es la dirección del recurso que se desea obtener o la ruta a la que se quiere enviar datos.

• opciones es un objeto que permite configurar la petición (método HTTP como GET, POST, PUT, etc., cabeceras, cuerpo de la petición, entre otros).

Ejemplo1: fetch

Si quisieramos realizar una solicitud de datos a un servicio externo que nos devuelva, por ejemplo, un listado de usuarios en formato JSON, podríamos hacer algo como lo siguiente:

```
fetch("https://jsonplaceholder.typicode.com/users")
  .then(response => {
    // Comprobamos si la respuesta es correcta (código HTTP 200-299)
    if (!response.ok) {
```

```
      throw new Error(`Error en la petición: ${response.status}`);
    }
    // Convertimos la respuesta en formato JSON
    return response.json();
  })
  .then(data => {
    console.log("Listado de usuarios:", data);
  })
  .catch(error => {
    console.error("Hubo un problema con la petición:", error);
  });
```

Flujo de ejecución:

1. Se llama a fetch pasandole la URL del recurso a consumir.

2. A través del objeto response se realizan dos acciones, la primera, si ha habido éxito al realizar la petición y, la segunda, el “parseo” de los datos obtenidos en formato JSON al formato nativo de JavaScript. Mientras se realiza la petición, el programa no se bloquea y continúa su ejecución.

3. Los datos ya en formato JavaScript se obtienen en el objeto data y se muestran por consola.

4. Si se produce un error (e.g., un problema de conexión o un código HTTP fuera de rango 2xx), se captura en el bloque catch.

Ejemplo 2: setTimeout + fetch

Este ejemplo realiza una solicitud HTTP para obtener datos de un servidor, pero antes de hacerlo esperamos un tiempo determinado, simulando un “retraso controlado” antes de disparar la petición.

```
console.log("Programa iniciado. Preparando para hacer fetch en 3 segundos...");

// Esperamos 3 segundos antes de hacer la petición
setTimeout(function() {
```

```
  console.log("Iniciando fetch...");
  fetch("https://jsonplaceholder.typicode.com/posts/1")
    .then(function(response) {
      if (!response.ok) {
        throw new Error(`Error en la solicitud: ${response.status}
`);
      }
      return response.json();
    })
    .then(function(data) {
      console.log("Datos recibidos (tras 3 segundos de retraso):",
data);
    })
    .catch(function(error) {
      console.error("Hubo un error en la petición:", error);
    });
}, 3000);
```

Explicación:

1. Se imprime un mensaje de inicio y se programa una función con setTimeout que se disparará a los 3 segundos.

2. Pasados esos 3 segundos, se ejecuta la función interna, que realiza un fetch a jsonplaceholder.typicode.com para obtener el detalle de un “post” con ID=1.

3. Una vez que los datos llegan, se muestran en la consola.

Ejemplo 3: setInterval + fetch

Aquí realizamos una consulta al servidor cada X segundos para actualizar cierta información, simulando un sondeo (polling). En este caso, cada 5 segundos se obtiene un recurso y se muestra por consola. Si la respuesta indica algún error, lo manejaremos. También añadimos la posibilidad de detener el sondeo con un botón en la página.

HTML:

```html
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>Ejemplo setInterval + fetch</title>
</head>
<body>
  <button id="btnDetener">Detener sondeo</button>
  <script src="app.js"></script>
</body>
</html>
```

JavaScript:

```javascript
const intervaloSondeo = setInterval(function() {
  console.log("Realizando petición al servidor...");
  fetch("https://jsonplaceholder.typicode.com/todos/1")
    .then(function(response) {
      if (!response.ok) {
        throw new Error(`Error en la solicitud: ${response.status}
`);
      }
      return response.json();
    })
    .then(function(data) {
      console.log("Datos obtenidos:", data);
    })
    .catch(function(error) {
      console.error("Error al obtener los datos:", error);
    });
}, 5000);
// Escuchamos el evento de clic en el botón para detener el sondeo
const btnDetener = document.getElementById("btnDetener");
```

```
btnDetener.addEventListener("click", function() {
  clearInterval(intervaloSondeo);
  console.log("Sondeo detenido.");
});
```

Conclusión

Como puedes observar, cada una de estas APIs se puede usar de forma individual, pero también se combinan de manera frecuente en proyectos reales para lograr comportamientos más complejos:

- setTimeout y setInterval se encargan de temporizar acciones, permitiendo programar ejecuciones retrasadas o periódicas.
- fetch es clave para realizar peticiones HTTP y manejar datos remotos sin bloquear el hilo principal.

Al unirlos, JavaScript adquiere la capacidad de realizar sondeos recurrentes a un servidor, programar acciones puntuales tras cierto tiempo o mostrar datos tras un periodo controlado de espera. Estos mecanismos asíncronos son fundamentales en el desarrollo moderno de aplicaciones web, pues permiten ofrecer experiencias dinámicas y reactivas a los usuarios.

ASINCRONISMO EN JAVASCRIPT

TEMA 06 – XHR Y AJAX

6.1. Introducción

A lo largo de la historia de JavaScript, uno de los elementos clave que ha revolucionado la comunicación asíncrona entre el cliente y el servidor es el concepto de **AJAX** (Asynchronous JavaScript and XML). AJAX permite actualizar partes de una página web sin recargarla completamente, lo que mejora significativamente la experiencia del usuario. En este contexto, la interfaz **XMLHttpRequest (XHR)**, ha sido una herramienta esencial para implementar esta funcionalidad. Aunque en la actualidad se emplean métodos más modernos como **fetch()**, XMLHttpRequest sigue siendo un componente fundamental para comprender los principios básicos del intercambio de datos en segundo plano, sin bloquear la interacción del usuario con la página web.

En este apartado, exploraremos qué es XMLHttpRequest, cómo se utiliza para realizar solicitudes HTTP (como GET o POST) y procesar datos en formato JSON o XML. Además, presentaremos un ejemplo práctico que ilustra su funcionamiento y, finalmente, compararemos brevemente sus características con las de la función fetch para destacar sus principales diferencias y ventajas.

6.1. Qué es XMLHttpRequest y cómo se usa para solicitudes HTTP

XMLHttpRequest (a menudo abreviado como XHR) es un objeto proporcionado por los navegadores web que permite realizar solicitudes HTTP y HTTPS de forma asíncrona. Su principal ventaja es la de habilitar la comunicación con el servidor sin necesidad de recargar toda la página, lo que se conoce comúnmente como **AJAX** (Asynchronous JavaScript and XML). Con XMLHttpRequest, podemos:

1. **Enviar y recibir datos en segundo plano** mientras el usuario sigue interactuando con la aplicación.

2. **Hacer peticiones a diferentes métodos HTTP** (GET, POST, PUT, DELETE, etc.).

3. **Trabajar con distintos formatos de datos**: JSON, XML, texto plano, HTML, etc.

4. **Personalizar cabeceras (headers)** en nuestras solicitudes y, de igual modo, leer las cabeceras de respuesta.

Antes de la llegada de fetch(), XMLHttpRequest era la forma estándar de realizar llamadas AJAX. Aun con la aparición de técnicas más modernas, es importante entenderlo en detalle ya que sigue siendo ampliamente utilizado en proyectos ya implantados.

Creación de una instancia de XMLHttpRequest

El primer paso para utilizar esta interfaz es crear una instancia del objeto:

```
let xhr = new XMLHttpRequest();
```

Una vez creada la instancia xhr, podemos usarla para preparar y enviar solicitudes a un servidor.

Solicitud con XMLHttpRequest

1. **Definir el tipo de solicitud y la URL**: se utiliza el método open() para indicar el método HTTP (GET, POST, etc.) y la dirección (URL) a la que vamos a enviar la petición.

```
xhr.open('GET', 'ruta/del/servidor', true);
```

El tercer parámetro indica si la petición será asíncrona (true) o síncrona (false). Prácticamente siempre desearemos trabajar de forma asíncrona.

2. **Configurar cabeceras y eventos** (opcional):

• Podemos ajustar cabeceras con setRequestHeader().

• Definir funciones callback o manejadores de eventos para controlar lo que sucederá cuando la respuesta llegue o en caso de error.

•Por ejemplo, podemos asignar una función al evento onreadystatechange o utilizar xhr.addEventListener('readystatechange', callback).

3. **Enviar la solicitud**: mediante el método send() se lanza la petición al servidor. Si utilizamos el método GET, normalmente no se envía cuerpo de la petición; en cambio, para POST se suele enviar un cuerpo (datos del formulario, JSON, etc.).

```
xhr.send();
```

4. **Recibir y procesar la respuesta**: cuando el servidor responde, podemos obtener su contenido a través de propiedades como xhr.responseText o xhr.responseXML, dependiendo del formato con que se haya devuelto la información.

El evento clave para detectar el momento en que la respuesta está lista es readystatechange. Dentro de este evento, comprobamos si xhr.readyState === 4 y xhr.status === 200, lo que indica que la respuesta se ha recibido con éxito.

• readyState puede tener los valores:

1. 0: no se ha inicializado (se acaba de crear el objeto).

2. 1: conexión establecida con open().

3. 2: se han recibido las cabeceras de la respuesta.

4. 3: está recibiendo el cuerpo de la respuesta.

5. 4: la respuesta ha sido recibida por completo.

• status típicamente es 200 si la solicitud se ha realizado con éxito, y variará en función de otros casos (404, 500, 401, etc.).

Ejemplo: Solicitud GET y manejo de datos JSON

Veamos un ejemplo práctico de cómo realizar una **solicitud GET** con XMLHttpRequest y procesar la respuesta, tanto en formato **JSON** como en **XML**. Este ejemplo puede servir como una plantilla, la cual posteriormente se

podrá adaptar a otros métodos como POST, PUT o DELETE, si fuera necesario.

Supongamos que en el servidor tenemos un endpoint que devuelve información en formato JSON, por ejemplo, en la ruta https://api.ejemplo.com/datos. El código para realizar la solicitud y manejar la respuesta JSON podría ser el siguiente:

```
// Creamos la instancia
let xhr = new XMLHttpRequest();
// Abrimos la solicitud indicando método y URL
xhr.open('GET', 'https://api.ejemplo.com/datos', true);
// Opcional: podemos establecer cabeceras si es necesario
// xhr.setRequestHeader('Content-Type', 'application/json');

// Definimos la función que se ejecutará cada vez que cambie el estado de la petición
xhr.onreadystatechange = function () {
  // Verificamos que haya finalizado la recepción de la respuesta
  if (xhr.readyState === 4) {
    // Comprobamos si la respuesta se recibió correctamente
    if (xhr.status === 200) {
      // Convertimos el texto de respuesta a un objeto JSON
      let datos = JSON.parse(xhr.responseText);
    // Aquí podemos manipular los datos según nuestras necesidades
      console.log('Datos recibidos (JSON):', datos);
    } else {
      // Manejo de error
        console.error('Error al realizar la solicitud. Código de estado:', xhr.status);
    }
  }
};
```

```
// Enviamos la solicitud
xhr.send();
```

Explicación:

1. **Creación**: new XMLHttpRequest().

2. **Configuración**: open('GET', 'https://api.ejemplo.com/datos', true) abre una petición GET a la URL especificada con modo asíncrono.

3. **Control del evento**: la función anónima asignada a onreadystatechange se ejecuta cada vez que el estado de la petición cambia.

4. **Validación de respuesta**: comprobamos si readyState === 4 (respuesta completa) y status === 200 (éxito).

5. **Procesamiento de datos**: empleamos JSON.parse para convertir el texto en un objeto JSON.

Ejemplo: Solicitud GET y manejo de datos XML

Si el servidor nos devuelve un documento en formato XML, podemos accederlo a través de la propiedad responseXML. Veamos un ejemplo:

```
let xhr = new XMLHttpRequest();
xhr.open('GET', 'https://api.ejemplo.com/datos.xml', true);
xhr.onreadystatechange = function () {
  if (xhr.readyState === 4) {
    if (xhr.status === 200) {
      let xmlDoc = xhr.responseXML;
      if (xmlDoc) {
        // Por ejemplo, si el nodo raíz se llama <raiz>
        let raiz = xmlDoc.getElementsByTagName('raiz')[0];
        console.log('Nodo raíz:', raiz.nodeName);
       // Podemos seguir explorando el DOM XML según sea necesario
      } else {
          console.error('No se pudo interpretar la respuesta como
XML.');
```

```
      }
    } else {
        console.error('Error al realizar la solicitud. Código de estado:', xhr.status);
    }
  }
};

xhr.send();
```

Explicación:

1. Una vez completada la petición, usamos xhr.responseXML para obtener un documento DOM que representa el archivo XML.

2. Podemos navegar el DOM resultante para extraer la información deseada mediante métodos como getElementsByTagName, getAttribute, etc.

3. En caso de que la respuesta no fuera realmente XML o hubiera algún problema, responseXML podría devolverse como null.

6.3. Comparación con fetch

XMLHttpRequest fue la tecnología pionera para realizar llamadas asíncronas en JavaScript. Sin embargo, a lo largo del tiempo apareció el método **fetch()**, que ofrece una **API más moderna** y basada en **promesas**, haciéndola más legible y permitiendo un uso más natural de funcionalidades recientes como async/await. He aquí una comparación:

1. **Sintaxis y estilo de programación**

• XMLHttpRequest: utiliza eventos (onreadystatechange) y callbacks para controlar la evolución de la solicitud. Esto puede llevar a estructuras de código algo más complejas, especialmente si se anidan varias peticiones.

• fetch: implementa promesas, por lo que facilita la composición de peticiones encadenadas, el manejo de errores con catch() y la legibilidad del código con la sintaxis async/await.

2. **Soporte y compatibilidad**

• Ambas son ampliamente soportadas por los navegadores modernos.

• XMLHttpRequest existe desde antes de fetch, lo que lo hace compatible con la gran mayoría de navegadores, incluyendo versiones más antiguas.

3. **Manejo de la respuesta**

• XMLHttpRequest: se accede mediante propiedades como responseText o responseXML.

• fetch: la respuesta se gestiona a través de métodos asíncronos como .json() o .text(), entre otros.

4. **Eventos y errores**

• XMLHttpRequest: propone distintos eventos (onload, onerror, onreadystatechange) para manejo de éxito, error y progreso.

• fetch: trabaja con promesas; el estado de la petición se maneja con .then() y .catch(). Los errores de red se capturan fácilmente con .catch(). Para manejar errores de estado (como 404, 500, etc.), se verifica manualmente la propiedad response.ok.

5. **Progresos de carga y otros detalles**

• XMLHttpRequest: resulta más sencillo monitorear el progreso de la carga con el evento onprogress.

• fetch: todavía no cuenta con un soporte nativo tan directo para seguimiento del progreso en la descarga del cuerpo de la respuesta (existen métodos basados en ReadableStream, pero la implementación y manipulación es más compleja).

6.4. Ejemplos

Ejemplo 1

En este primer fragmento de código se demuestra lo esencial: crear una instancia de **XMLHttpRequest**, abrir una conexión GET a una URL y capturar el resultado.

```
// EJEMPLO 1: Uso básico de XMLHttpRequest para una solicitud GET (texto plano o HTML)
function cargarContenidoTexto() {
  // 1. Crear instancia de XMLHttpRequest
  let xhr = new XMLHttpRequest();
  // 2. Configurar la petición (método y URL)
  xhr.open('GET', 'https://ejemplo.com/contenido.txt', true);
  // 3. Manejadores de evento: onreadystatechange o onload
  xhr.onreadystatechange = function () {
    // Verificar si la respuesta ha llegado (readyState = 4) y es correcta (status = 200)
    if (xhr.readyState === 4 && xhr.status === 200) {
      // 4. Obtener el texto de la respuesta
      let respuesta = xhr.responseText;
     // 5. Mostrar la respuesta en consola o manipularla en el DOM
      console.log('Contenido recibido:', respuesta);
    }
  };
  // 6. Enviar la solicitud
  xhr.send();
}
// Llamada a la función
cargarContenidoTexto();
```

Explicación

• Creación de la instancia con new XMLHttpRequest().

• Uso de open('GET', 'url', true) para una petición asíncrona.

• Validación de readyState === 4 y status === 200 para asegurarnos de que la respuesta sea exitosa.

• Obtención de la respuesta con xhr.responseText (útil cuando el servidor devuelve texto plano o HTML).

Ejemplo 2

Este segundo ejemplo muestra cómo recuperar datos JSON desde un endpoint y convertirlos a un objeto JavaScript manipulable.

```
// EJEMPLO 2: Solicitud GET y manejo de JSON
function cargarDatosJSON() {
  // 1. Crear la instancia
  let xhr = new XMLHttpRequest();
  // 2. Definir la URL que devuelve datos en formato JSON
  xhr.open('GET', 'https://api.ejemplo.com/datos', true);
  // 3. Manejador de eventos
  xhr.onreadystatechange = function () {
    if (xhr.readyState === 4) {
      if (xhr.status === 200) {
        // 4. Transformar el texto de la respuesta en un objeto JavaScript
        let datos = JSON.parse(xhr.responseText);
        // 5. Manipular los datos (ejemplo: mostrarlos en consola)
        console.log('Datos JSON recibidos:', datos);
      } else {
        // Manejo de error si la respuesta no es 200
        console.error('Error al cargar datos JSON. Código de estado:', xhr.status);
      }
    }
  };
  // 6. Enviar la solicitud
  xhr.send();
```

```
}
// Llamada a la función
cargarDatosJSON();
```

Explicación

• El uso de JSON.parse(xhr.responseText) para convertir de texto plano a objeto JavaScript.

• El control de errores en caso de no obtener un código de estado 200.

Ejemplo 3

Este fragmento de código muestra cómo se trabajan datos en formato XML y se accede a su contenido como un DOM XML.

```
// EJEMPLO 3: Solicitud GET y manejo de XML
function cargarDatosXML() {
  // 1. Crear la instancia
  let xhr = new XMLHttpRequest();
  // 2. Definir la URL que devuelve un documento XML
  xhr.open('GET', 'https://api.ejemplo.com/datos.xml', true);
  // 3. Evento para manejar la respuesta
  xhr.onreadystatechange = function () {
    if (xhr.readyState === 4) {
      if (xhr.status === 200) {
        // 4. Recibimos el DOM XML
        let xmlDoc = xhr.responseXML;
        // 5. Manipular el DOM XML (ejemplo: leer etiquetas
"usuario")
        if (xmlDoc) {
          let usuarios = xmlDoc.getElementsByTagName('usuario');
          for (let i = 0; i < usuarios.length; i++) {
            console.log('Usuario #' + (i + 1) + ':',
usuarios[i].textContent);
          }
```

```
        } else {
          console.error('No se pudo interpretar la respuesta como XML.');
        }
      } else {
        console.error('Error al cargar datos XML. Código de estado:', xhr.status);
      }
    }
  };
  // 6. Enviar la solicitud
  xhr.send();
}
// Llamada a la función
cargarDatosXML();
```

Explicación

- El uso de xhr.responseXML para obtener un objeto DOM.
- La posterior manipulación con métodos del DOM (por ejemplo, getElementsByTagName).

Ejemplo 4

En este ejemplo, veremos cómo sería la **misma** solicitud GET implementada con **fetch()** en comparación a **XMLHttpRequest**, para que el alumnado entienda las diferencias sintácticas y de manejo de promesas.

```
// EJEMPLO 4A: Petición GET con XMLHttpRequest
function cargarConXHR() {
  let xhr = new XMLHttpRequest();
  xhr.open('GET', 'https://api.ejemplo.com/datos', true);
  xhr.onreadystatechange = function () {
    if (xhr.readyState === 4 && xhr.status === 200) {
      console.log('XHR:', JSON.parse(xhr.responseText));
```

```
    }
  };
  xhr.send();
}
// EJEMPLO 4B: Petición GET con fetch
async function cargarConFetch() {
  try {
    let respuesta = await fetch('https://api.ejemplo.com/datos');
    if (respuesta.ok) {
      let datos = await respuesta.json();
      console.log('fetch:', datos);
    } else {
      console.error('Error con fetch. Código de estado:',
respuesta.status);
    }
  } catch (error) {
    console.error('Error de red u otro problema:', error);
  }
}
// Llamadas a las funciones
cargarConXHR();
cargarConFetch();
```

Diferencias clave

1. **XHR** usa readyState y callbacks (en este caso onreadystatechange) para detectar cuándo la respuesta está lista.

2. **fetch** se basa en promesas, lo que permite una sintaxis más limpia con async/await.

3. En el caso de fetch, debemos manejar el error de estado manualmente verificando respuesta.ok.

Ejemplo 5

Para completar la visión de cómo funciona XMLHttpRequest, aquí se muestra un ejemplo de envío de datos (una solicitud POST) con un objeto JSON:

```
// EJEMPLO 5: Petición POST con envío de JSON
function enviarDatosJSON() {
  let xhr = new XMLHttpRequest();
  xhr.open('POST', 'https://api.ejemplo.com/guardar', true);
  // Cabecera para indicar que enviamos JSON
  xhr.setRequestHeader('Content-Type', 'application/json;charset=UTF-8');
  // Manejar la respuesta
  xhr.onreadystatechange = function () {
    if (xhr.readyState === 4) {
      if (xhr.status === 200) {
        console.log('Datos enviados correctamente:', xhr.responseText);
      } else {
        console.error('Error al enviar datos. Código de estado:', xhr.status);
      }
    }
  };
  // Objeto JavaScript con la información que vamos a enviar
  let data = {
    nombre: 'Juan',
    edad: 30,
    ocupacion: 'Desarrollador',
  };
  // Convertimos a JSON y lo enviamos
  xhr.send(JSON.stringify(data));
}
// Llamada a la función
enviarDatosJSON();
```

Explicación:

• Uso de POST y setRequestHeader para indicar el tipo de contenido.

• JSON.stringify para transformar el objeto JavaScript a JSON antes de enviarlo.

Ejemplo 6: Obtener un recurso en JSON y, a continuación, enviar un formulario por POST (XHR)

1. Primero hacemos una petición GET para obtener datos en JSON y mostramos esa información.

2. Luego realizamos una **petición POST** para enviar un formulario (también en JSON), aprovechando que el usuario podría haber rellenado algún campo y se quiere guardar en el servidor.

```
function flujoCompletoXHR() {
  // 1. Petición GET para datos JSON
  let xhrGet = new XMLHttpRequest();
     xhrGet.open('GET',   'https://api.ejemplo.com/datosIniciales',
true);
  xhrGet.onreadystatechange = function () {
    if (xhrGet.readyState === 4) {
      if (xhrGet.status === 200) {
        let datosRecibidos = JSON.parse(xhrGet.responseText);
        console.log('Datos iniciales:', datosRecibidos);
// 2. Simular un formulario que completamos y enviamos por POST
        let formulario = {
          nombre: 'María',
          curso: 'Desarrollo Web',
        };
        // 3. Crear nueva instancia para la petición POST
        let xhrPost = new XMLHttpRequest();
                                                xhrPost.open('POST',
'https://api.ejemplo.com/guardarFormulario', true);
```

```
                        xhrPost.setRequestHeader('Content-Type',
'application/json;charset=UTF-8');
        xhrPost.onreadystatechange = function () {
          if (xhrPost.readyState === 4) {
            if (xhrPost.status === 200) {
                  console.log('Formulario guardado correctamente:',
xhrPost.responseText);
            } else {
                console.error('Error al guardar formulario. Código
de estado:', xhrPost.status);
            }
          }
        };
        // 4. Enviamos el formulario
        xhrPost.send(JSON.stringify(formulario));
      } else {
         console.error('Error al obtener datos iniciales. Código de
estado:', xhrGet.status);
      }
    }
  };
  xhrGet.send();
}
// Invocamos la función
flujoCompletoXHR();
```

En este **flujo** se muestra el manejo secuencial de peticiones: primero un **GET** y luego un **POST**, todo ello usando XMLHttpRequest. Se encadenan las operaciones en el onreadystatechange de la primera llamada, asegurándonos de no enviar el POST hasta que la respuesta GET haya sido satisfactoria.

Ejemplo 7: Comparar en una sola función un proceso con XHR y otro con fetch

Este último ejemplo integra lo siguiente:

1. Realiza una petición **GET** con **XMLHttpRequest** para obtener datos en JSON.

2. Realiza una petición **GET** con **fetch()** para otra URL distinta, comparando la forma de capturar y procesar esos datos.

```
async function procesoMixtoXHRyFetch() {
  // --- Parte A: Obtener datos con XMLHttpRequest ---
  const xhr = new XMLHttpRequest();
  xhr.open('GET', 'https://api.ejemplo.com/datosXHR', true);
  xhr.onreadystatechange = function () {
    if (xhr.readyState === 4 && xhr.status === 200) {
      let datosXHR = JSON.parse(xhr.responseText);
      console.log('Datos obtenidos con XHR:', datosXHR);
    }
  };
  xhr.send();
  // --- Parte B: Obtener datos con fetch ---
  try {
    let respuesta = await fetch('https://api.ejemplo.com/datosFetch');
    if (respuesta.ok) {
      let datosFetch = await respuesta.json();
      console.log('Datos obtenidos con fetch:', datosFetch);
    } else {
      console.error('Error fetch. Código de estado:', respuesta.status);
    }
  } catch (error) {
    console.error('Error de red u otro problema con fetch:', error);
  }
}
// Llamada a la función para ejecutar ambos procesos
procesoMixtoXHRyFetch();
```

En este caso, tanto la parte XHR como la parte fetch se ejecutan de forma independiente dentro de una misma función, ilustrando la coexistencia de ambas soluciones. Podríamos incluso usar los resultados de la primera (XHR) para condicionar el fetch, pero en este ejemplo se muestran como peticiones paralelas que no dependen una de la otra.

6.5. Conclusión

Si bien fetch es la opción recomendada para la mayoría de proyectos nuevos por su estilo asíncrono más moderno y limpio, entender a fondo XMLHttpRequest es importante para mantener proyectos antiguos (legacy), asegurarse compatibilidad con versiones de navegadores más viejas y comprender los fundamentos históricos de la comunicación asíncrona en JavaScript.

En resumen, **XMLHttpRequest** sigue siendo una pieza clave del ecosistema JS. A lo largo de los años, la comunidad ha ido evolucionando hacia nuevas interfaces como fetch, pero **XHR** conserva su relevancia tanto en términos de compatibilidad como para escenarios donde se necesite un control más bajo nivel de ciertos aspectos (por ejemplo, seguimiento de progreso con eventos). Conocer ambas opciones permite elegir la mejor herramienta para cada situación y mantener o modernizar proyectos según sea necesario.

ASINCRONISMO EN JAVASCRIPT

TEMA 07 – PROCESAMIENTO ASÍNCRONO CON XML Y JSON

7.1. Introducción

En el ámbito del desarrollo web, especialmente cuando trabajamos en el entorno de JavaScript, es fundamental comprender a fondo cómo se gestionan los datos en distintos formatos y de qué manera podemos intercambiarlos de forma asíncrona entre el cliente y el servidor. Dos formatos muy utilizados para la transferencia de datos son XML y JSON.

Aunque JSON ha ido ganando más popularidad, especialmente debido a su integración más sencilla con JavaScript, es conveniente conocer las ventajas y aplicaciones de ambos.

A continuación, abordaremos tres puntos esenciales que ayudarán a consolidar el conocimiento sobre el uso de XML y JSON en procesos asíncronos:

1. Diferencias entre XML y JSON.
2. Cómo trabajar con JSON usando la API `fetch`.
3. Introducción básica al manejo de XML con `DOMParser`.

Con ello, pretendemos que tengas una visión clara de estos dos formatos, sepas utilizarlos en aplicaciones web reales y apliques buenas prácticas a la hora de desarrollar tus proyectos.

7.2. Diferencias entre XML y JSON

Estructura y sintaxis

- **XML (Extensible Markup Language)**
    - Está basado en etiquetas (similar al HTML, pero sin etiquetas predefinidas).
    - Cada bloque de datos se encierra en etiquetas de apertura y cierre.
    - Requiere una sintaxis estricta y un cierre correcto de cada etiqueta.

- Puede incluir atributos en las etiquetas.
- Es más “verbose” a la hora de representar información, ya que cada elemento requiere sus etiquetas correspondientes.

```
<persona>
  <nombre>Juan</nombre>
  <edad>25</edad>
</persona>
```

- **JSON (JavaScript Object Notation)**
  - Está basado en la notación de objetos de JavaScript.
  - Utiliza llaves `{}` para definir objetos y corchetes `[]` para definir listas o arreglos.
  - Los datos se representan en pares `clave: valor`, separados por comas.
  - Requiere un uso correcto de comillas y comas, especialmente en las claves y cadenas de texto.
  - Es menos “verbose”, lo que facilita su lectura y escritura en la mayoría de los casos.

```
{
  "nombre": "Juan",
  "edad": 25
}
```

Legibilidad y popularidad

- **XML** era extremadamente popular hace años en el intercambio de datos entre sistemas. Su principal fortaleza residía en la flexibilidad y la posibilidad de definir esquemas para validar la estructura (XSD). Sin embargo, la complejidad en la escritura y lectura lo ha hecho menos práctico en entornos de desarrollo web más modernos.
- **JSON** se ha convertido en el estándar de facto para la transferencia de datos en aplicaciones web (por ejemplo, en APIs REST). Es mucho más ligero y está directamente integrado en JavaScript, ya que la sintaxis coincide con la definición de objetos en este lenguaje. Asimismo, su popularidad se debe a la facilidad de serializar y deserializar datos (convertir de objeto a cadena y viceversa).

Soporte en JavaScript

- **XML** puede requerir bibliotecas o APIs concretas para su manipulación, como *`DOMParser`* y *`XMLSerializer`*, o la propia interfaz *`XMLHttpRequest`* de antaño si nos remontamos a las primeras implementaciones del AJAX clásico.
- **JSON** se maneja de forma nativa en JavaScript a través de los métodos *`JSON.stringify()`* (para convertir un objeto en una cadena JSON) y *`JSON.parse()`* (para convertir una cadena JSON en un objeto JavaScript).

Uso actual

- **XML** se suele utilizar en casos muy específicos (por ejemplo, la manipulación de ficheros de configuración, ciertas arquitecturas empresariales más antiguas, o la lectura de ficheros SVG, que internamente están basados en XML, aunque no siempre se gestione de la misma manera que un archivo XML puro).
- **JSON** es la opción preferida para casi cualquier tipo de comunicación con APIs modernas, así como para almacenar configuraciones, ya que ofrece mayor ligereza y una integración fluida con JavaScript.

En conclusión, la elección entre XML y JSON dependerá del contexto, de las necesidades del proyecto y de los requerimientos de compatibilidad o de validación. Sin embargo, hoy en día, JSON es la alternativa más habitual y recomendada para la mayoría de desarrollos web en JavaScript.

7.3. Cómo trabajar con JSON

Una de las formas más comunes de realizar peticiones asíncronas para obtener datos en JSON es a través de la API *`fetch`*. Esta API nos permite, de manera nativa en el navegador, realizar solicitudes HTTP y procesar las respuestas de forma relativamente sencilla y moderna, sustituyendo el antiguo uso de *`XMLHttpRequest`*.

Estructura básica de una petición con `fetch`

```
fetch('https://api.ejemplo.com/datos')
  .then(response => {
```

```
    // Comprobamos si la respuesta es correcta
    if (!response.ok) {
                throw new Error('Error en la respuesta: ' +
response.status);
    }
    // parseamos la respuesta a JSON
    return response.json();
  })
  .then(data => {
    // Aquí manejamos el objeto JS resultante
    console.log(data);
  })
  .catch(error => {
    // Manejamos cualquier error durante la petición
    console.error('Ha ocurrido un error:', error);
  });
```

Primero, llamamos a *`fetch`* con la URL o endpoint que queramos.

- Después, la promesa resultante contiene un objeto Response que necesitamos verificar con *`response.ok`*.
- A continuación, utilizamos el método *`response.json()`* para convertir la respuesta a un objeto JavaScript (antes era una cadena).
- Una vez que tenemos el objeto, podemos trabajar con él directamente, por ejemplo, para mostrar datos en una página o para procesarlos de la forma que necesitemos.

Fetch con async/await

```
async function obtenerDatos() {
  try {
    const response = await fetch('https://api.ejemplo.com/datos');
    if (!response.ok) {
      throw new Error('Error en la respuesta: ' +
response.status);
    }
```

```
    const data = await response.json();
    console.log(data);
  } catch (error) {
    console.error('Ha ocurrido un error:', error);
  }
}
obtenerDatos();
```

- *async* se declara en la función que va a utilizar *await*.
- *await* “pausa” la ejecución hasta que la promesa se resuelva, evitando la necesidad de múltiples *.then*.

Buenas prácticas al trabajar con JSON

- Validar siempre que las respuestas del servidor sean correctas, tanto por código de estado (*status*) como por el contenido retornado.
- Estructurar el JSON de manera clara y limpia para facilitar la lectura y el mantenimiento del código.
- Manejar adecuadamente los errores (por ejemplo, si el JSON no está bien formado o si la petición no devuelve lo esperado).
- Usar HTTPS en la medida de lo posible para proteger la transferencia de datos.

Con estos puntos, se pretende mostrar que la combinación de JSON y *fetch* facilita sobremanera la comunicación con servidores y la gestión de datos asíncronos, posicionando a JSON como la opción más cómoda en proyectos JavaScript modernos.

7.4. Introducción al manejo de XML con DOMParser

Aunque JSON sea el formato más común, sigue habiendo situaciones en las que se trabaja con XML. Para ello, JavaScript proporciona herramientas nativas que permiten parsear el XML y transformarlo en un objeto manipulable.

¿Qué es DOMParser?

DOMParser es un objeto disponible en la mayoría de los navegadores modernos, que convierte cadenas de texto que contienen marcado XML (o HTML) en un documento DOM que se puede manipular con las mismas funciones que se utilizan para manipular el DOM del documento HTML de la página.

Uso básico de DOMParser

Imaginemos que recibimos un *string* con contenido XML. Para parsearlo, podemos hacer lo siguiente:

```
const xmlString = `
  <catalog>
    <book id="1">
      <title>Aprendiendo JavaScript</title>
      <author>Juan Pérez</author>
    </book>
    <book id="2">
      <title>Profundizando en XML</title>
      <author>María García</author>
    </book>
  </catalog>
`;

// Creamos una instancia de DOMParser
const parser = new DOMParser();
// Parseamos la cadena con el formato "application/xml" o
"text/xml"
const xmlDoc = parser.parseFromString(xmlString, "text/xml");

// Ahora xmlDoc es un documento DOM con nodos accesibles
const catalog = xmlDoc.getElementsByTagName('catalog')[0];
const books = catalog.getElementsByTagName('book');

for (let i = 0; i < books.length; i++) {
```

```
  const title = books[i].getElementsByTagName('title')
     [0].textContent;
  const author = books[i].getElementsByTagName('author')
     [0].textContent;
  console.log(`Libro ${i + 1}: ${title}, de ${author}`);
}
```

Se crea una instancia de DOMParser.

- Se utiliza *parseFromString*, indicando el tipo de contenido.
- El resultado es un documento DOM con métodos como *getElementsByTagName()*, *querySelectorAll()*, etc.
- De esta forma, podemos recorrer el árbol XML y obtener la información de cada elemento.

Conversión inversa con XMLSerializer

Si necesitamos convertir el DOM en una cadena de texto XML (por ejemplo, tras haber hecho modificaciones sobre el DOM resultante), podemos usar *XMLSerializer* de la siguiente manera:

```
const serializer = new XMLSerializer();
const nuevoXMLString = serializer.serializeToString(xmlDoc);
console.log(nuevoXMLString);
```

Este proceso nos permite, por ejemplo, actualizar nodos o atributos en un documento XML y luego generar de nuevo la cadena resultante para enviarla a un servidor o guardarla en un fichero.

- **Consideraciones al trabajar con XML**
- Es importante asegurarse de que el XML sea válido (etiquetas bien anidadas, atributos correctamente definidos, etc.). De lo contrario, *DOMParser* puede generar errores o ignorar partes del documento.
- A diferencia de JSON, que se integra más naturalmente con objetos JavaScript, el trabajo con XML implica la manipulación del DOM, lo que puede resultar menos intuitivo para desarrolladores no familiarizados con la estructura en forma de árbol.

7.5. Ejemplos

Ejemplo 1: Estudiante y asignaturas que cursa.

El formato JSON podría ser:

```
{
  "nombre": "Carla",
  "edad": 22,
  "matriculada": true,
  "asignaturas": ["Programación", "Bases de Datos", "Entornos de Desarrollo"]
}
```

- **Clave-valor (nombre, edad, matriculada)**: Cada propiedad se expresa como un par *`"clave": valor`*.
- *`"nombre"`* y *`"edad"`* son propiedades del objeto, con valores “Carla” (tipo cadena) y 22 (tipo numérico), respectivamente.
- *`"matriculada"`* es una propiedad booleana (true/false).
- **Array de asignaturas**: La clave *`"asignaturas"`* contiene un array delimitado por corchetes *`[ ... ]`* y compuesto por valores de tipo *`string`*.

Para procesar este JSON en JavaScript, usaríamos, por ejemplo, el método *`JSON.parse()`* si se encuentra en forma de cadena, o podríamos recibirlo ya convertido a objeto gracias a la API *`fetch`*:

```
const jsonString = `
{
  "nombre": "Carla",
  "edad": 22,
  "matriculada": true,
  "asignaturas": ["Programación", "Bases de Datos", "Entornos de Desarrollo"]
}
```

```
`;

// Parseamos la cadena para convertirla en un objeto JavaScript

const estudiante = JSON.parse(jsonString);

console.log(estudiante.nombre); // "Carla"

console.log(estudiante.asignaturas[0]); // "Programación"
```

Una vez parseado, `estudiante` se convierte en un objeto normal de JavaScript: *`estudiante.nombre`* accedería al valor *`"Carla"`* y *`estudiante.asignaturas`* daría acceso al array de asignaturas.

La correspondecia en XML sería:

```
<estudiante>
  <nombre>Carla</nombre>
  <edad>22</edad>
  <matriculada>true</matriculada>
  <asignaturas>
    <asignatura>Programación</asignatura>
    <asignatura>Bases de Datos</asignatura>
    <asignatura>Entornos de Desarrollo</asignatura>
  </asignaturas>
</estudiante>
```

Parseo en JavaScript con DOMParser

```
const xmlString = `
<estudiante>
  <nombre>Carla</nombre>
  <edad>22</edad>
```

```
  <matriculada>true</matriculada>
  <asignaturas>
    <asignatura>Programación</asignatura>
    <asignatura>Bases de Datos</asignatura>
    <asignatura>Entornos de Desarrollo</asignatura>
  </asignaturas>
</estudiante>
`;
// Creación de un DOMParser
const parser = new DOMParser();
// Parseamos el string como XML
const xmlDoc = parser.parseFromString(xmlString, "text/xml");
// Acceso al contenido
const nombre = xmlDoc.getElementsByTagName("nombre")[0].textContent;
const edad = xmlDoc.getElementsByTagName("edad")[0].textContent;
const matriculada = xmlDoc.getElementsByTagName("matriculada")[0].textContent;
// Para el array de asignaturas:
const asignaturas = xmlDoc.getElementsByTagName("asignatura");
console.log("Nombre:", nombre);           // "Carla"
console.log("Edad:", edad);               // "22"
console.log("Matriculada:", matriculada); // "true"
for (let i = 0; i < asignaturas.length; i++) {
  console.log(`Asignatura ${i + 1}:`, asignaturas[i].textContent);
}
```

En este ejemplo, cada nodo *<asignatura>* se maneja de forma individual. No hay un array nativo, sino una colección de nodos que podemos recorrer con un bucle.

Ejemplo 2: Estructuras anidadas con objetos y arrays

En este segundo ejemplo, representaremos la información de un curso con varios módulos y alumnos inscritos. Aquí se apreciará la anidación de objetos y arrays dentro de un mismo JSON:

```json
{
  "curso": "Desarrollo de Aplicaciones Web",
  "codigo": "DAW2024",
  "modulos": [
    {
      "nombre": "Lenguaje de Marcas",
      "numHoras": 120,
      "aprobados": [
        { "alumno": "Lucía", "nota": 8.5 },
        { "alumno": "Pedro", "nota": 7.0 }
      ]
    },
    {
      "nombre": "Entornos de Desarrollo",
      "numHoras": 100,
      "aprobados": [
        { "alumno": "Ana", "nota": 9.0 },
        { "alumno": "Carlos", "nota": 6.8 }
      ]
    }
  ]
}
```

Objeto principal: Contiene las claves *"curso"*, *"codigo"* y *"modulos"*.

- **Array de módulos**: *"modulos"* es un array en el que cada elemento es un objeto con sus propias propiedades (nombre, numHoras...) y un array anidado llamado "aprobados".
- **Array "aprobados"**: Cada elemento de este array es un objeto con dos propiedades: *"alumno"* y *"nota"*.

Manipulación en JavaScript

Una vez que tenemos este JSON (por ejemplo, al recibirlo de un servidor), podríamos trabajar con él como sigue:

```
const jsonString2 = `
{
  "curso": "Desarrollo de Aplicaciones Web",
  "codigo": "DAW2024",
  "modulos": [
    {
      "nombre": "Lenguaje de Marcas",
      "numHoras": 120,
      "aprobados": [
        { "alumno": "Lucía", "nota": 8.5 },
        { "alumno": "Pedro", "nota": 7.0 }
      ]
    },
    {
      "nombre": "Entornos de Desarrollo",
      "numHoras": 100,
      "aprobados": [
        { "alumno": "Ana", "nota": 9.0 },
        { "alumno": "Carlos", "nota": 6.8 }
      ]
```

```
    }
  ]
}
`;
const datosCurso = JSON.parse(jsonString2);
console.log("Nombre del curso:", datosCurso.curso);
console.log("Código:", datosCurso.codigo);

datosCurso.modulos.forEach((modulo, index) => {
  console.log(`\nMódulo ${index + 1}:`, modulo.nombre);
  console.log(`Horas: ${modulo.numHoras}`);

  modulo.aprobados.forEach(aprobado => {
    console.log(`- Alumno: ${aprobado.alumno}, Nota: ${aprobado.nota}`);
  });
});
```

datosCurso.modulos es un array de objetos.

- Cada objeto del array *modulos* tiene su propio array aprobados.
- Navegamos cada nivel con JavaScript de manera sencilla, pues JSON se traduce “uno a uno” a la estructura interna de objetos y arrays de JavaScript.

Correspondencia en XML

Si quisiéramos describir la misma información en **XML**, podríamos diseñar la siguiente estructura:

```
<curso>
  <nombre>Desarrollo de Aplicaciones Web</nombre>
  <codigo>DAW2024</codigo>
  <modulos>
    <modulo>
```

```
      <nombre>Lenguaje de Marcas</nombre>
      <numHoras>120</numHoras>
      <aprobados>
        <aprobado>
          <alumno>Lucía</alumno>
          <nota>8.5</nota>
        </aprobado>
        <aprobado>
          <alumno>Pedro</alumno>
          <nota>7.0</nota>
        </aprobado>
      </aprobados>
    </modulo>
    <modulo>
      <nombre>Entornos de Desarrollo</nombre>
      <numHoras>100</numHoras>
      <aprobados>
        <aprobado>
          <alumno>Ana</alumno>
          <nota>9.0</nota>
        </aprobado>
        <aprobado>
          <alumno>Carlos</alumno>
          <nota>6.8</nota>
        </aprobado>
      </aprobados>
    </modulo>
  </modulos>
</curso>
```

- **Elemento** `<curso>`: Sirve de contenedor principal.

- **Estructuras anidadas**:
  - `<modulos>` se compone de varios elementos `<modulo>`.
  - Cada `<modulo>` dispone de sus elementos `<nombre>`, `<numHoras>` y `<aprobados>`.
  - `<aprobados>` contiene múltiples elementos `<aprobado>`, uno por cada alumno con su correspondiente `<alumno>` y `<nota>`.
- No hay arrays nativos, pero la repetición de elementos `<modulo>` y `<aprobado>` funciona como una forma de lista o conjunto de elementos.

Parseo en JavaScript

Usamos la misma idea con **DOMParser**:

```
const xmlString2 = `
<curso>
  <nombre>Desarrollo de Aplicaciones Web</nombre>
  <codigo>DAW2024</codigo>
  <modulos>
    <modulo>
      <nombre>Lenguaje de Marcas</nombre>
      <numHoras>120</numHoras>
      <aprobados>
        <aprobado>
          <alumno>Lucía</alumno>
          <nota>8.5</nota>
        </aprobado>
        <aprobado>
          <alumno>Pedro</alumno>
          <nota>7.0</nota>
        </aprobado>
      </aprobados>
```

```
    </modulo>
    <modulo>
      <nombre>Entornos de Desarrollo</nombre>
      <numHoras>100</numHoras>
      <aprobados>
        <aprobado>
          <alumno>Ana</alumno>
          <nota>9.0</nota>
        </aprobado>
        <aprobado>
          <alumno>Carlos</alumno>
          <nota>6.8</nota>
        </aprobado>
      </aprobados>
    </modulo>
  </modulos>
</curso>
`;
const parser = new DOMParser();
const doc = parser.parseFromString(xmlString2, "text/xml");

const nombreCurso = doc.getElementsByTagName("nombre")[0].textContent;
const codigoCurso = doc.getElementsByTagName("codigo")[0].textContent;

console.log("Curso:", nombreCurso);
console.log("Código:", codigoCurso);

const modulos = doc.getElementsByTagName("modulo");
```

```
for (let i = 0; i < modulos.length; i++) {
  const modulo = modulos[i];
  const nombreModulo = modulo.getElementsByTagName("nombre")[0].textContent;
  const numHoras = modulo.getElementsByTagName("numHoras")[0].textContent;

  console.log(`\nMódulo ${i + 1}: ${nombreModulo}`);
  console.log(`Horas: ${numHoras}`);

  const aprobados = modulo.getElementsByTagName("aprobado");
  for (let j = 0; j < aprobados.length; j++) {
    const aprobado = aprobados[j];
    const alumno = aprobado.getElementsByTagName("alumno")[0].textContent;
    const nota = aprobado.getElementsByTagName("nota")[0].textContent;
    console.log(`- Alumno: ${alumno}, Nota: ${nota}`);
  }
}
```

Ejemplo 2: Petición JSON y parseo XML local

En este ejemplo, primero realizamos una llamada para obtener datos en formato JSON (simulando una lista de películas), y después parseamos una cadena XML local (simulando catálogos distintos), todo en una misma función.

```
async function obtenerDatosYParsearXML() {
  try {
    // 1. Obtenemos datos en JSON desde un endpoint ficticio
    const respuestaJSON = await fetch('https://api.ejemplo.com/peliculas');
    if (!respuestaJSON.ok) {
      throw new Error('Error al obtener las películas: ' + respuestaJSON.status);
```

```
    }
    const peliculas = await respuestaJSON.json();
    console.log('Películas en JSON:', peliculas);
    // 2. Parseamos una cadena XML local
    const xmlString = `
      <catalogo>
        <entrada>
          <nombre>Entrada General</nombre>
          <precio>10</precio>
        </entrada>
        <entrada>
          <nombre>Entrada VIP</nombre>
          <precio>25</precio>
        </entrada>
      </catalogo>
    `;
    const parser = new DOMParser();
    const xmlDoc = parser.parseFromString(xmlString, 'application/xml');
    const entradas = xmlDoc.getElementsByTagName('entrada');
   // Recorremos las entradas y mostramos la información
    for (let i = 0; i < entradas.length; i++) {
      const nombre = entradas[i].getElementsByTagName('nombre')[0].textContent;
      const precio = entradas[i].getElementsByTagName('precio')[0].textContent;
      console.log(`Entrada ${i + 1}: ${nombre}, precio: ${precio}`);
    }
  } catch (error) {
    console.error('Error en el proceso combinado:', error);
```

```
  }

}

// Llamada a la función

obtenerDatosYParsearXML();
```

Ejemplo 3: construcción de JSON y XML dinámicamente

Vamos a crear un objeto JSON y un documento XML en tiempo de ejecución. Después, simulamos un envío (por consola) de ambos formatos, como si se fuesen a mandar al servidor.

```
function generarYMostrarDatos() {

  // 1. Generar un objeto JSON dinámicamente

  const usuario = {

    nombre: 'Ana',

    edad: 30,

    tecnologias: ['JavaScript', 'Node.js', 'React']

  };

  // Convertimos el objeto en una cadena JSON

  const usuarioJSON = JSON.stringify(usuario);

  console.log('Objeto JSON generado:\n', usuarioJSON);

  // 2. Generar un documento XML dinámicamente

  const docXML = document.implementation.createDocument('', '', null);

  const raiz = docXML.createElement('usuario');

  // Creamos nodos para nombre y edad

  const nombreNodo = docXML.createElement('nombre');

  nombreNodo.textContent = 'Ana';

  const edadNodo = docXML.createElement('edad');

  edadNodo.textContent = '30';

  // Creamos un nodo "tecnologias" con varios hijos
```

```
const tecnologiasNodo = docXML.createElement('tecnologias');
const tecnologia1 = docXML.createElement('tecnologia');
tecnologia1.textContent = 'JavaScript';
const tecnologia2 = docXML.createElement('tecnologia');
tecnologia2.textContent = 'Node.js';
const tecnologia3 = docXML.createElement('tecnologia');
tecnologia3.textContent = 'React';
// Añadimos las tecnologías al nodo "tecnologias"
tecnologiasNodo.appendChild(tecnologia1);
tecnologiasNodo.appendChild(tecnologia2);
tecnologiasNodo.appendChild(tecnologia3);
// Unimos nombre, edad y tecnologias a la raíz
raiz.appendChild(nombreNodo);
raiz.appendChild(edadNodo);
raiz.appendChild(tecnologiasNodo);
// Añadimos la raíz al documento
docXML.appendChild(raiz);
// Serializamos
const serializer = new XMLSerializer();
const usuarioXML = serializer.serializeToString(docXML);
console.log('Documento XML generado:\n', usuarioXML);
// Envío de datos JSON al servidor,
fetch('https://api.ejemplo.com/subir-datos', {
method: 'POST',
   headers: {
     'Content-Type': 'application/json' // En caso de mandar JSON
   },
   body: usuarioJSON
 })
```

```
    .then(res => res.json())
    .then(resultado => console.log('Respuesta del servidor:', resultado))
    .catch(error => console.error('Error al enviar datos JSON:', error));
  // Envío de datos XML al servidor,
   fetch('https://api.ejemplo.com/subir-xml', {
    method: 'POST',
    headers: {
      'Content-Type': 'application/xml' // En caso de mandar XML
    },
    body: usuarioXML
   })
    .then(res => res.text())
     .then(resultado => console.log('Respuesta del servidor al XML:',
resultado))
    .catch(error => console.error('Error al enviar datos XML:', error));
}
// Ejecutamos la función
generarYMostrarDatos();
```

Estos ejemplos muestran cómo, en JavaScript**,** puedes:

- **Recibir o enviar datos en formato JSON** usando *`fetch`*.
- **Parsear y manipular XML** usando el objeto *`DOMParser`* y, si lo necesitas, volver a serializar con *`XMLSerializer`*.
- Combinar ambos formatos en un mismo script, ya sea porque tu aplicación consume datos JSON y mantiene configuraciones en XML, o por otros motivos de compatibilidad y requerimientos de un proyecto.

Recuerda que:

- **JSON** se integra de manera más natural con JavaScript y es el estándar predominante para APIs modernas.
- **XML** sigue siendo útil en contextos específicos (sistemas legacy, configuraciones con esquemas formales, ficheros SVG, etc.).

- Con `fetch`, `async`/`await` y `DOMParser`, dispones de herramientas nativas y potentes para cubrir la mayoría de escenarios de comunicación y procesamiento de datos asíncronos.

Estos ejemplos constituyen la base para manejar de forma profesional e independiente las peticiones asíncronas y la gestión de datos en cualquier aplicación web real desarrollada en JavaScript.

ASINCRONISMO EN JAVASCRIPT

TEMA 08 – PROMESAS

8.1. Introducción

En el desarrollo moderno con JavaScript, especialmente en el entorno del navegador y en Node.js, uno de los retos más habituales es la gestión de operaciones asíncronas. Anteriormente, se utilizaban callbacks para manejar estas operaciones, lo cual podía derivar en un “callback hell” o XHR con una estructura sintáctica con cierta complejidad. Con la introducción de las *Promises* a partir de 2015, se ofreció una solución más clara y manejable para escribir, organizar y razonar sobre código asíncrono.

A continuación, profundizaremos en la base teórica de las Promesas, sus estados y sus métodos fundamentales. Además, mostraremos ejemplos simples para ilustrar su uso.

8.2. Promesas

Una *Promise* es un objeto que representa un valor que puede estar disponible ahora, en el futuro o nunca. Técnicamente, una Promesa modela un flujo asíncrono de forma que, cuando se ejecuta una operación que implica cierta latencia (lectura de archivos, consultas a bases de datos, llamadas a APIs, entre otras), podamos tratar el resultado (o el posible error) sin necesidad de bloquear el resto de la ejecución.

Definición de una Promesa

Podemos crear una nueva promesa mediante la clase nativa Promise de JavaScript, utilizando su constructor:

```
const miPromesa = new Promise((resolve, reject) => {
  // Aquí va la operación asíncrona o un bloque de código
});
```

El constructor `Promise` recibe una función "*ejecutora*" que, a su vez, recibe dos parámetros: `resolve` y `reject`.

- *resolve(valor)* se invoca cuando la operación asíncrona ha tenido éxito o ha finalizado de forma satisfactoria, donde *valor* es el resultado que la promesa entrega.
- *reject(error)* se invoca cuando la operación ha fallado o se produce algún tipo de error, donde *error* describe la causa del fallo.

Estados de una Promesa

Las promesas en JavaScript cuentan con tres estados principales:

1. **Pending (pendiente)**: El estado inicial, antes de que se haya cumplido o rechazado.
2. **Fulfilled (cumplida)**: Significa que la promesa ha terminado de forma satisfactoria, es decir, la función *resolve* ha sido llamada con éxito.
3. **Rejected (rechazada)**: Indica que la promesa no pudo completarse correctamente y se invocó la función *reject*.

Una vez que la promesa cambia de estado a *fulfilled* o *rejected*, se considera que está "*settled*" (resuelta, en el sentido de que ya no está pendiente). Sin embargo, debemos tener presente que, una vez *settled*, el estado no puede volver a cambiar.

Para visualizarlo de una forma más clara:

1. **Pending** → pasa a → **Fulfilled**
2. **Pending** → pasa a → **Rejected**

Después de esto, la promesa se considera establecida (settled) y ya no cambiará más.

8.2. Métodos básicos: .then, .catch y .finally

Las promesas se consumen (o gestionan) mediante métodos encadenables que nos permiten procesar el resultado o el error de forma ordenada.

.then()

El método *.then()* se encarga de procesar el resultado exitoso (estado *fulfilled*) de una promesa. La sintaxis básica es la siguiente:

```
miPromesa.then( (resultado) => { // Manejo de la respuesta, si todo salió bien
  }
);
```

El callback que se pasa a *.then()* se ejecuta únicamente cuando la promesa ha sido resuelta de manera satisfactoria (se ha llamado a *resolve*). Además, *.then()* puede devolver otra promesa, lo que permite encadenar múltiples operaciones asíncronas de manera más clara:

```
miPromesa
  .then((resultado) => {
    // primer bloque de manejo
    return otroValor; // puede ser un valor, o incluso otra promesa
  })
  .then((resultado2) => {
    // bloque de manejo del resultado devuelto por el .then anterior
  });
```

.catch()

El método *.catch()* se utiliza para capturar y manejar las situaciones en las que la promesa es rechazada (estado *rejected*). Es decir, cuando ocurre un error y se llama a la función *reject*:

```
miPromesa.catch(
  (error) => {
    // Manejo de errores
    console.error("Ha ocurrido un error:", error);
  }
);
```

En combinación con *.then()*, *.catch()* nos ofrece una forma clara de separar la lógica de éxito de la de error, incrementando la legibilidad:

```
miPromesa
  .then((resultado) => {
    // Si todo va bien
    console.log("Resultado:", resultado);
  })
  .catch((error) => {
    // Si algo falla
    console.error("Se ha producido un error:", error);
  });
```

Por otro lado, si dentro de un *.then()* ocurre un error de ejecución (por ejemplo, una excepción no controlada), este error también será

“encapsulado” en la promesa y podrá ser recogido más adelante por un *.catch()* posterior.

.finally()

El método *.finally()* se ejecuta siempre, independientemente de si la promesa ha sido cumplida o rechazada. Su principal utilidad radica en operaciones de limpieza o en pasos finales que deseamos realizar sin importar el resultado:

```
miPromesa
  .then((resultado) => {
    console.log("Resultado:", resultado);
  })
  .catch((error) => {
    console.error("Error en la promesa:", error);
  })
  .finally(() => {
    console.log("Esta parte se ejecutará siempre");
  });
```

8.3. EJEMPLOS

EJEMPLO 1: Funcionamiento de una Promise

Supongamos que deseamos simular una operación asíncrona que tarde cierto tiempo (como podría ser una petición a una API). Para ello, utilizaremos *setTimeout*, que nos permitirá ejecutar código tras un retardo determinado:

```
function simularOperacionAsincrona() {
  return new Promise((resolve, reject) => {
    console.log("Iniciando operación asíncrona...");
    setTimeout(() => {
      const exito = true; // Cambiar a false para probar el caso de error
      if (exito) {
        resolve("Operación completada con éxito.");
      } else {
        reject("Ocurrió un error durante la operación.");
```

```
      }
    }, 2000); // 2 segundos de retardo
  });
}


simularOperacionAsincrona()
  .then((mensaje) => {
    console.log("Mensaje de éxito:", mensaje);
  })
  .catch((error) => {
    console.error("Mensaje de error:", error);
  })
  .finally(() => {
    console.log("Fin de la operación asíncrona.");
  });
```

1. Se define la función *`simularOperacionAsincrona()`*, que retorna una nueva promesa.
2. En el *`setTimeout`*, se determina a través de la variable *`exito`* si la promesa se resuelve correctamente (*`resolve`*) o se rechaza (*`reject`*).
3. Al llamar a *`simularOperacionAsincrona()`*, utilizamos *`.then()`* para manejar el éxito, *`.catch()`* para manejar el error y *`.finally()`* para cualquier paso final (por ejemplo, registro en consola).

EJEMPLO 2: Encadenamiento de Promesas

En muchos casos, deberemos realizar varias operaciones asíncronas secuenciales, donde el resultado de una sirva como entrada a la siguiente. Con promesas, podemos hacerlo de manera clara y ordenada:

```
function obtenerUsuario() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      const usuario = { id: 1, nombre: "Juan Pérez" };
      // Suponemos que la obtención del usuario fue exitosa
      resolve(usuario);
    }, 1000);
  });
}
```

```
function obtenerPedidosPorUsuario(usuarioId) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      // Simulamos que se obtienen pedidos del usuario
      const pedidos = [
        { pedidoId: 101, usuarioId: usuarioId, producto: "Libro" },
        { pedidoId: 102, usuarioId: usuarioId, producto: "Camiseta" },
      ];
      resolve(pedidos);
    }, 1500);
  });
}

// Encadenamiento
obtenerUsuario()
  .then((usuario) => {
    console.log("Usuario obtenido:", usuario);
    return obtenerPedidosPorUsuario(usuario.id);
  })
  .then((pedidos) => {
    console.log("Pedidos del usuario:", pedidos);
  })
  .catch((error) => {
    console.error("Error en la cadena de promesas:", error);
  })
  .finally(() => {
    console.log("Fin de la ejecución.");
  });
```

En este ejemplo:

1. Primero obtenemos los datos de un usuario mediante la función *`obtenerUsuario()`*.
2. Con el resultado exitoso de la promesa anterior, llamamos a *`obtenerPedidosPorUsuario(usuario.id)`*.

3. Si en algún punto se produce un error (por ejemplo, si *obtenerUsuario()* o *obtenerPedidosPorUsuario()* llama a `reject`), la ejecución pasa de inmediato al método *.catch()*.
4. Finalmente, con *.finally()* cerramos el flujo con una notificación final.

EJEMPLO 3: Promesa estado fullfilled y rejected

En este apartado, mostraremos cómo se produce la transición entre los estados de *pending* (pendiente), *fulfilled* (cumplida) y *rejected* (rechazada). Para ello, crearemos promesas sencillas que, transcurrido un tiempo, se resuelven satisfactoriamente o se rechazan.

Promesa *fulfilled*

```
function promesaFulfilled() {
  return new Promise((resolve, reject) => {
    console.log("Promesa en estado PENDING...");

    // Simulamos una operación asíncrona
    setTimeout(() => {
      console.log("Operación completada con éxito.");
      // Llamamos a resolve para indicar que la promesa pasa a estado FULFILLED
      resolve("Esta promesa ha sido resuelta satisfactoriamente.");
    }, 2000);
  });
}
// Consumimos la promesa
promesaFulfilled()
  .then((resultado) => {
    console.log("Estado Fulfilled:", resultado);
  })
  .catch((error) => {
    // Este bloque no se ejecutará en este ejemplo
    console.error("Estado Rejected:", error);
  });
```

En este ejemplo:

1. Se crea la promesa y se informa que inicialmente está en *pending*.

2. Pasados 2 segundos, se llama a *resolve*, lo cual la lleva a *fulfilled*.
3. Al consumirse la promesa con *.then()*, se muestra por consola el resultado de éxito.

Promesa *rejected*

```
function promesaRejected() {
  return new Promise((resolve, reject) => {
    console.log("Promesa en estado PENDING...");
    // Simulamos una operación asíncrona
    setTimeout(() => {
      console.log("Operación fallida.");
      // Llamamos a reject para indicar que la promesa pasa a estado REJECTED
      reject("La promesa ha sido rechazada debido a un error.");
    }, 2000);
  });
}
// Consumimos la promesa
promesaRejected()
  .then((resultado) => {
    // Este bloque no se ejecutará en este ejemplo
    console.log("Estado Fulfilled:", resultado);
  })
  .catch((error) => {
    console.error("Estado Rejected:", error);
  });
```

En este caso:

1. Se inicia la promesa en estado *pending*.
2. Tras un retardo, se llama a *reject*, lo que provoca el estado *rejected*.
3. El *.then()* no se ejecuta, y el flujo pasa a *.catch()*, donde se gestionará el error.

EJEMPLO 4: Encadenamiento de promesas y majeno independiente de errores

En este ejemplo, mezclamos promesas que funcionan bien con otras que podrían fallar, y mostramos cómo *.catch()* puede aparecer en diferentes partes de la cadena, o agruparse en uno solo al final.

```
function validarToken() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      const tokenValido = Math.random() > 0.3; // 70% de probabilidad de éxito
      if (tokenValido) {
        resolve("Token válido.");
      } else {
        reject("Token inválido, no autorizado.");
      }
    }, 800);
  });
}

function solicitarDatosPrivados() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      // Simulamos obtención de datos
      resolve({ informacion: "Datos privados del usuario." });
    }, 1200);
  });
}

function procesarDatos(datos) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      // Simulamos un posible error en el procesamiento
      const procesoExitoso = Math.random() > 0.5;
      if (procesoExitoso) {
        resolve(`Datos procesados con éxito: ${JSON.stringify(datos)}`);
      } else {
        reject("Hubo un problema al procesar los datos.");
      }
    }, 1000);
```

```
  });
}
// Cadena de Promesas
validarToken()
  .then((mensajeToken) => {
    console.log(mensajeToken);
    // Si la validación fue exitosa, solicitamos datos privados
    return solicitarDatosPrivados();
  })
  .then((datosPrivados) => {
    console.log("Datos privados obtenidos:", datosPrivados);
    // Encadenamos el procesamiento de datos
    return procesarDatos(datosPrivados);
  })
  .then((resultadoProcesado) => {
    console.log(resultadoProcesado);
  })
  .catch((error) => {
    // Cualquier error que ocurra en la cadena es capturado aquí
    console.error("Ha ocurrido un error en alguna parte del proceso:", error);
  })
  .finally(() => {
    console.log("Proceso de validación y procesamiento finalizado.");
  });
```

Observaciones:

1. *`validarToken()`* puede fallar si el token no es válido (rechaza la promesa).
2. *`solicitarDatosPrivados()`* no falla en este ejemplo (siempre resuelve).
3. *`procesarDatos()`* puede fallar aleatoriamente al procesar los datos (rechaza la promesa).
4. Cualquier rechazo en la cadena se intercepta en el *`.catch()`*.
5. *`.finally()`* muestra el mensaje final incondicionalmente.

8.4.Conclusiones

Las Promesas en JavaScript suponen un gran avance con respecto al modelo tradicional basado exclusivamente en *callbacks*. Su flujo de trabajo permite escribir código asíncrono legible y fácil de mantener, reduciendo el riesgo de “callback hell” y ofreciendo un método más claro para encadenar y manejar diferentes etapas de un proceso asíncrono.

Dominar los estados de las promesas y los métodos `.then()`, `.catch()` y `.finally()` es un paso imprescindible para comprender el funcionamiento de JavaScript en profundidad, especialmente en aplicaciones web modernas y en servicios backend con Node.js. A partir de estas bases, se puede construir conocimiento sólido sobre otras herramientas asíncronas, como `async/await`, que internamente también hacen uso de las promesas.

En resumen, las promesas permiten:

1. Ejecutar tareas asíncronas de forma no bloqueante.
2. Manejar adecuadamente distintos resultados (éxito o error).
3. Encadenar operaciones asíncronas con facilidad.
4. Mejorar la legibilidad y mantenimiento del código.

ASINCRONISMO EN JAVASCRIPT

TEMA 09 – ASYNC / AWAIT

9.1. Introducción

Para situarnos en contexto, es importante matizar por qué `async/await` existe en JavaScript y en qué se basa. Cuando hablamos de programación asíncrona en JavaScript, hay varios momentos clave:

1. **Callbacks**: fueron la forma tradicional de manejar operaciones asíncronas. Sin embargo, a medida que los programas crecían en complejidad, se hacía habitual encontrarnos con estructuras de "callback hell", anidando funciones y generando código difícil de leer y de mantener.
2. **Promesas**: introducidas para solucionar los problemas de las funciones de callback anidadas. Las Promesas permiten encadenar métodos como `then()` y `catch()`, lo cual aporta un mayor orden y manejabilidad a las operaciones asíncronas. Aun así, en proyectos grandes, una larga sucesión de `then()` encadenados puede llegar a ser confusa y costosa de leer.
3. **Async/Await**: surge como una evolución o *azúcar sintáctico* sobre las Promesas, incorporado formalmente a partir de ECMAScript 2017. El término ***azúcar sintáctico*** significa que no se trata de un mecanismo nuevo, sino que internamente sigue basándose en el comportamiento de las Promesas, pero ofrece una forma de escribir el código más clara, parecida a la programación secuencial tradicional.

En pocas palabras, `async/await` no reemplaza a las Promesas, sino que las utiliza para manejar los procesos asíncronos. Gracias a esto, el código asíncrono se aproxima más al estilo sincrónico de siempre (un paso tras otro, en orden), lo que favorece la legibilidad y reduce los posibles errores lógicos.

¿Por qué async/await se considera más legible?

- ***Menor anidación***: los bloques `then()` se sustituyen por un código que se ve más secuencial.
- ***Manejo de errores más coherente***: en lugar de depender de métodos específicos como `.catch()`, se utiliza `try/catch` nativo de JavaScript, que resulta más familiar y consistente.
- ***Lectura lineal***: la palabra clave `await` "pausa" la ejecución de la función hasta que la Promesa se resuelve (o se rechaza), por lo que el flujo se lee de manera lineal, como si fuese código síncrono.

De esta forma, `async/await` es como una capa sobre las Promesas, ya que internamente no deja de usarlas, pero nos ofrece un modo más sencillo e intuitivo de trabajar con ellas.

9.2. Uso de async/await

Para empezar a usar `async/await`, debemos entender claramente las dos palabras clave involucradas: **async** y **await**.

- `async`: se antepone a la declaración de una función (por ejemplo, `async function miFuncion() { ... }`), indicando que dicha función va a contener código asíncrono y que, además, retornará automáticamente una Promesa.
- `await`: solo puede usarse dentro de una función declarada como `async`. Su función principal es "esperar" a que una Promesa se resuelva o se rechace, como si pausara la ejecución hasta que el resultado esté disponible. Durante ese tiempo de espera, la ejecución del resto del código dentro de esa función queda en suspenso. Eso sí, mientras tanto, JavaScript puede continuar ejecutando otros trozos de código en el programa principal (event loop), lo que mantiene el carácter asíncrono del lenguaje.

```
// Simulando la obtención de datos de un servidor con un retardo usando setTimeout
function obtenerDatosServidor() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      // Imaginemos que los datos llegan correctamente
      const datos = { usuario: "Juan", id: 123 };
      resolve(datos);
    }, 2000); // 2 segundos de retardo
  });
}

async function procesarDatos() {
  console.log("Iniciando solicitud de datos...");
  // Esperamos a que la Promesa se resuelva
  const datos = await obtenerDatosServidor();
  console.log("Datos recibidos:", datos);
  console.log("Continuamos con el flujo de ejecución...");
}

procesarDatos();
// Salida en consola:
// "Iniciando solicitud de datos..."
// (Tras 2 segundos) "Datos recibidos: { usuario: 'Juan', id: 123 }"
// "Continuamos con el flujo de ejecución..."
```

Observa que `await` no bloquea todo el programa, sino solo el bloque de código dentro de la función `async`. Fuera de esa función, el resto del código JavaScript puede continuar ejecutándose (por ejemplo, eventos del navegador, otras funciones, etc.). Sin embargo, dentro de la función `procesarDatos()`, la línea con `await obtenerDatosServidor()` actúa como un “alto” hasta que la Promesa se cumpla o se rechace.

Este paradigma hace que el código sea más lineal y fácil de entender en comparación con el encadenamiento de `then()` y `catch()`, o el uso intensivo de callbacks.

9.3. Manejo de errores con try/catch

Cuando usábamos Promesas con el enfoque clásico, disponíamos del método `.catch()` para capturar los errores. Con `async/await`, se adopta la sintaxis clásica de JavaScript para gestionar excepciones: `try/catch`.

La idea es envolver las llamadas asíncronas que podrían fallar en un bloque `try/catch`, capturando así cualquier error que la Promesa lance en caso de que sea rechazada.

```
async function obtenerUsuario() {
  // Supongamos que esta función retorna una promesa con datos de un usuario
  return { nombre: "Pedro", edad: 30 };
}

async function mostrarUsuario() {
  try {
    // Si la promesa falla, se lanza un error que será capturado por el catch
    const usuario = await obtenerUsuario();
    console.log("Usuario obtenido:", usuario);
  } catch (error) {
    console.error("Ocurrió un error al obtener el usuario:", error);
  }
}
mostrarUsuario();
```

Si `obtenerUsuario()` rechazara la Promesa (por ejemplo, usando `reject` en la lógica interna), el bloque `catch` dentro de `mostrarUsuario()` recibiría el error y podríamos manejarlo apropiadamente.

```
function obtenerDetallesProducto(idProducto) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      // Imaginemos que si el idProducto es negativo o cero, es un ID inválido
      if (idProducto > 0) {
```

```
      resolve({ id: idProducto, nombre: "Teclado", precio: 30 });
    } else {
      reject(new Error("ID de producto inválido"));
    }
  }, 1000);
 });
}

async function procesarProducto(idProducto) {
 try {
  const producto = await obtenerDetallesProducto(idProducto);
  console.log("Detalles del producto:", producto);
 } catch (error) {
  console.error("Error al procesar producto:", error.message);
 }
}

procesarProducto(10); // Caso válido
procesarProducto(-1); // Caso inválido
```

En el caso válido, tras un segundo se imprime:

```
Detalles del producto: { id: 10, nombre: "Teclado", precio: 30 }
```

En el caso inválido, el mensaje de error se capturará y mostrará en consola:

```
Error al procesar producto: ID de producto inválido
```

Gracias a `try/catch`, el tratamiento de errores en `async/await` se asemeja a lo que se haría en un lenguaje de programación tradicional. Esto reduce la necesidad de anidar múltiples `.catch()` en diferentes puntos, lo que agiliza la lectura y hace el flujo de trabajo más claro.

9.4. EJEMPLOS

EJEMPLO 1

Este primer ejemplo muestra cómo se podría escribir una función asíncrona con Promesas y luego la misma función usando `async/await`, para apreciar que `async/await` es, en esencia, *azúcar sintáctico* sobre las Promesas.

```
// Función que simula la obtención de datos de un servidor
function obtenerDatosConPromesa() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      const datos = { usuario: "Carlos", edad: 25 };
      // Simulamos que la operación se resuelve correctamente
      resolve(datos);
    }, 1000);
  });
}

// Uso de la función
obtenerDatosConPromesa()
  .then((resultado) => {
    console.log("Versión con Promesas - Datos recibidos:", resultado);
  })
  .catch((error) => {
    console.error("Versión con Promesas - Error:", error);
  });
```

Versión con Async/Await

```
// Función que simula la obtención de datos de un servidor, versión async/await
async function obtenerDatosAsyncAwait() {
  // Internamente, esta función sigue usando Promesas, pero se escribe de forma más legible
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      const datos = { usuario: "Carlos", edad: 25 };
      resolve(datos);
```

```
    }, 1000);
  });
}

async function mostrarDatos() {
  try {
    const resultado = await obtenerDatosAsyncAwait();
    console.log("Versión con Async/Await - Datos recibidos:", resultado);
  } catch (error) {
    console.error("Versión con Async/Await - Error:", error);
  }
}

// Llamamos a la función principal para ver el resultado en consola
mostrarDatos();
```

En la primera parte, se emplea `.then()` y `.catch()` para gestionar la Promesa. En la segunda, se usa `await` y un bloque `try/catch`. Ambos métodos hacen lo mismo, pero la versión `async/await` suele leerse de manera más secuencial.

EJEMPLO 2

Para este ejemplo, tenemos una función que simula la consulta de un listado de productos.

```
// Función que simula la obtención de un listado de productos
function obtenerProductos() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      const productos = [
        { id: 1, nombre: "Teclado", precio: 30 },
        { id: 2, nombre: "Ratón", precio: 15 },
        { id: 3, nombre: "Monitor", precio: 150 }
      ];
      resolve(productos);
    }, 1500);
```

```
  });
}

async function listarProductos() {
  console.log("Solicitando productos...");

  // El flujo de esta función se “detiene” aquí hasta que se resuelve la Promesa
  const lista = await obtenerProductos();

  console.log("Productos recibidos:");
  lista.forEach((producto) => {
    console.log(`- ${producto.nombre} (ID: ${producto.id}): ${producto.precio}€`);
  });

  console.log("Continuando la ejecución tras haber recibido la lista...");
}

// Llamamos a la función asíncrona
listarProductos();
```

EJEMPLO 3

En este ejemplo, veremos cómo capturar los errores de una Promesa rechazada dentro de una función marcada como *async*.

```
// Función que simula obtener datos de un usuario, pudiendo fallar
function obtenerUsuarioPorId(id) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      // Simulamos un fallo si el ID es <= 0
      if (id > 0) {
        resolve({ id: id, nombre: "María", rol: "Admin" });
      } else {
        reject(new Error("ID de usuario inválido"));
      }
    }, 1000);
  });
```

```
}

async function mostrarUsuario(id) {
  try {
    const usuario = await obtenerUsuarioPorId(id);
    console.log("Usuario obtenido:", usuario);
  } catch (error) {
    // Si la Promesa se rechaza, se captura aquí
    console.error("Error al obtener el usuario:", error.message);
  }
}

// Probar con un ID válido
mostrarUsuario(2);

// Probar con un ID inválido (lanzará la excepción en consola)
mostrarUsuario(0);
```

En este caso, se emplea el bloque `try/catch` para atrapar cualquier error que se produzca dentro de la función asíncrona. Si la Promesa se rechaza, el flujo salta al bloque `catch`, donde podemos manejar el error de la forma que necesitemos (mostrar un mensaje al usuario, reintentar la operación, etc.).

EJEMPLO 4

Supongamos que tenemos dos operaciones asíncronas: una para obtener la información de un usuario y otra para obtener sus pedidos (o compras) de una tienda en línea. Queremos unirlo todo de forma clara y legible:

```
// Función que obtiene información de un usuario
function obtenerInfoUsuario(idUsuario) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      if (idUsuario > 0) {
        resolve({ id: idUsuario, nombre: "Sofía" });
```

```
    } else {
      reject(new Error("ID de usuario no válido"));
    }
  }, 1000);
 });
}

// Función que obtiene la lista de pedidos de un usuario
function obtenerPedidosDeUsuario(idUsuario) {
 return new Promise((resolve, reject) => {
  setTimeout(() => {
    // Imaginamos que para este usuario existe una lista de pedidos
    const pedidos = [
      { numero: 101, total: 56 },
      { numero: 102, total: 120 },
    ];
    // Si no hay errores, resolvemos
    resolve(pedidos);
  }, 1200);
 });
}

async function mostrarDatosCompletosUsuario(idUsuario) {
 try {
  console.log("Iniciando obtención de datos...");

  // 1. Obtenemos la info básica del usuario
  const usuario = await obtenerInfoUsuario(idUsuario);
  console.log("Usuario encontrado:", usuario);

  // 2. Obtenemos los pedidos asociados a ese usuario
  const pedidos = await obtenerPedidosDeUsuario(usuario.id);
  console.log("Pedidos del usuario:", pedidos);

  console.log("Proceso finalizado con éxito.");
 } catch (error) {
```

```
    // Capturamos cualquier error de las dos funciones
    console.error("Error en la obtención de datos:", error.message);
  }
}

// Llamamos con un ID correcto
mostrarDatosCompletosUsuario(5);

// Llamamos con un ID incorrecto (<= 0), provoca error
mostrarDatosCompletosUsuario(-1);
```

En este ejemplo, se observa cómo:

1. La estructura secuencial (primero obtener el usuario, después los pedidos, etc.) es más fácil de entender.
2. El manejo de errores centralizado se realiza con `try/catch`, por lo que si falla la primera o la segunda llamada, el `catch` "atrapa" la excepción.

EJEMPLO 5

En este ejemplo, además de hacer dos solicitudes asíncronas, aprovechamos la sintaxis de `Promise.all()` junto a `async/await`. Esta combinación puede resultar útil cuando dos (o más) operaciones se pueden ejecutar en paralelo y queremos esperar a que todas finalicen antes de continuar.

```
// Función que simula obtener información de un curso
function obtenerInfoCurso(idCurso) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      if (idCurso > 0) {
        resolve({ id: idCurso, titulo: "Curso de JavaScript Avanzado" });
      } else {
        reject(new Error("ID de curso inválido"));
      }
    }, 1000);
  });
}
```

```
// Función que simula obtener el listado de alumnos de un curso
function obtenerAlumnosDelCurso(idCurso) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      if (idCurso > 0) {
        resolve(["Ana", "Luis", "Marta", "Pablo"]);
      } else {
        reject(new Error("No se pudieron obtener alumnos para este curso"));
      }
    }, 1500);
  });
}

async function cargarDatosCurso(idCurso) {
  try {
    console.log("Iniciando carga de datos del curso...");

    // Hacemos ambas llamadas en paralelo para ganar eficiencia
    const [curso, alumnos] = await Promise.all([
      obtenerInfoCurso(idCurso),
      obtenerAlumnosDelCurso(idCurso),
    ]);

    // Mostramos la información recibida
    console.log("Información del curso:", curso);
    console.log("Listado de alumnos:", alumnos);

    console.log("Datos del curso cargados con éxito.");
  } catch (error) {
    // Si ocurre algún error en cualquiera de las dos operaciones, se captura aquí
    console.error("Error al cargar los datos del curso:", error.message);
  }
}

// Llamada con un ID válido
```

```
cargarDatosCurso(10);


// Llamada con un ID inválido
cargarDatosCurso(0);
```

En este ejemplo:

1. Hemos utilizado `Promise.all()` para ejecutar ambas promesas en paralelo.
2. Gracias a `await`, esperamos a que ambas finalicen antes de asignar los resultados a las variables `curso` y `alumnos`.
3. Cualquier error que ocurra en una de las dos promesas se captura en el bloque `catch`.

9.5. CONCLUSIÓN

La llegada de `async/await` a JavaScript supuso un hito a la hora de trabajar con código asíncrono, simplificando la legibilidad y el manejo de excepciones. Aunque parezca una novedad radical, internamente se trata de una forma más “dulce” (o *azúcar sintáctico*) de usar Promesas. El resultado es un código más lineal y, por tanto, más fácil de mantener y depurar.

A modo de resumen, conviene retener los siguientes puntos clave:

1. **Async/Await se basa en Promesas**: no las elimina ni las sustituye, simplemente hace su uso más sencillo.
2. **Lectura secuencial**: el uso de `await` en una función `async` promueve que el código se lea como si fuese sincrónico.
3. **Manejo de errores**: mediante `try/catch` podemos capturar fácilmente los posibles errores que surjan en las operaciones asíncronas.
4. Comprender `async/await` puede ser más rápido que adaptarse a varios niveles de `then` y `callbacks`, ya que se aproxima más a la forma de pensar de la programación secuencial.

ASINCRONISMO EN JAVASCRIPT

TEMA 10 – WEB WORKERS

10.1. Introducción a los Web Workers

Hasta ahora hemos tratado el asincronismo desde la perspectiva clásica de JavaScript, es decir, un único hilo de ejecución junto con un segundo plano al que se envían las tareas asíncronas. HTML5 introdujo los **Web Workers**, cuyo objetivo principal es permitir la ejecución de scripts de manera **concurrente**, es decir, en hilo de ejecución separados y, de esta forma, liberar al hilo principal de la carga pesada, haciendo que la experiencia de usuario sea más fluida.

A continuación, veremos en detalle qué son los Web Workers, cómo crearlos y utilizarlos, y finalmente cuáles son sus limitaciones y buenas prácticas para sacar el máximo provecho de ellos.

10.2. Qué son y para qué sirven los Web Workers

Un Web Worker es un script de JavaScript que se ejecuta en un hilo distinto al principal con el objetivo de realizar tareas complejas o prolongadas. De esta manera:

- **El hilo principal** se encarga de gestionar la interfaz de la aplicación, la interacción con el DOM y la respuesta a eventos, manteniendo la aplicación receptiva en todo momento.
- **Los Web Workers** ejecutan operaciones intensivas de CPU sin interrumpir el flujo de la interfaz gráfica. Un ejemplo muy común es cuando necesitamos realizar cálculos matemáticos complejos, manipular grandes volúmenes de datos o procesar ficheros pesados.

La gran ventaja de los Web Workers es su capacidad de ejecutar código en paralelo. Cada Web Worker tiene su propio contexto global y no

comparte el objeto window del hilo principal, lo cual fomenta la independencia entre el cómputo intensivo y las funciones de actualización de la UI.

10.3. Diferencias entre asincronismo (Event Loop) y paralelismo (Web Workers)

Para entender mejor la importancia de los Web Workers, hay que distinguir dos conceptos clave:

1. **Asincronismo mediante el Event Loop:**

JavaScript, por naturaleza, es un lenguaje de ejecución **monohilo**. Todas las operaciones que realiza en el hilo principal se orquestan a través del **Event Loop**. En este modelo, las tareas se encolan (por ejemplo, con setTimeout, setInterval, peticiones AJAX, *promises*, etc.) y se ejecutan secuencialmente cuando el hilo principal está disponible. Las funciones asíncronas permiten que la interfaz de usuario no se bloquee mientras esperamos la resolución de procesos de E/S (entrada/salida), como peticiones de red, pero **no** aprovechan varios núcleos de CPU para ejecutar tareas en paralelo, sino que siguen gestionándose en un único hilo principal.

2. **Paralelismo con Web Workers:**

Los **Workers** permiten correr código de JavaScript en hilos separados del hilo principal, haciendo uso de varios núcleos de la CPU si están disponibles en el sistema. Esto significa que si una tarea compleja se está ejecutando en un Worker, el hilo principal puede seguir respondiendo a eventos del usuario, actualizar la interfaz, o realizar otras operaciones. En pocas palabras, con los Web Workers se consigue un verdadero **paralelismo**, siempre dentro de los límites del navegador y del modelo de hilos que ofrece el sistema operativo.

10.4. Creación de un Web Worker

La creación y uso de un Web Worker es relativamente sencilla, pero requiere prestar atención a algunos detalles específicos en cuanto a la comunicación entre el hilo principal y el propio Web Worker.

Comunicación entre el hilo principal y los Workers

La comunicación entre un Web Worker y el hilo principal se lleva a cabo a través de **mensajes**, usando los métodos y eventos:

1. `.postMessage()`

   Permite enviar información desde el hilo principal al Web Worker y viceversa.

2. `.onmessage()`

   Es un manejador de eventos que se dispara cuando uno de los dos extremos recibe un mensaje. Permite capturar y procesar la información que llega.

Estructura básica en el hilo principal

```
// 1. Creación de la instancia del Worker a partir de un archivo JS externo
const worker = new Worker('miWorker.js');
// 2. Envío de mensajes al Worker
worker.postMessage('Hola, Worker!');
// 3. Recepción de mensajes desde el Worker
worker.onmessage = function (event) {
  console.log('Mensaje recibido desde Worker:', event.data);
};
```

Estructura básica Web Worker (por ejemplo, miWorker.js)

```
// 1. Recepción de mensajes desde el hilo principal
onmessage = function (event) {
  console.log('Mensaje recibido desde el hilo principal:', event.data);
  // 2. Podemos procesar la información y enviar la respuesta
```

```
  const respuesta = `He recibido tu mensaje: ${event.data}`;
  postMessage(respuesta);
};
```

El intercambio de datos se basa en un mecanismo de **paso de mensajes**. No existe un acceso directo desde el Worker al objeto window o al DOM del hilo principal, lo cual es una de las restricciones más notables, pero también forma parte de la filosofía de aislar procesos y evitar bloqueos de la interfaz.

10.5. Limitaciones y buenas prácticas

Aunque los Web Workers son muy útiles, no están exentos de limitaciones. Para aprovecharlos de manera óptima, conviene tener presentes ciertos aspectos:

1. **Acceso limitado al DOM**

Dentro de un Worker no tienes acceso directo al DOM ni al objeto document o window. Esta limitación existe para evitar bloqueos en la interfaz y garantizar la independencia del hilo de ejecución. Si necesitas actualizar la interfaz, debes comunicarte con el hilo principal usando postMessage y dejar que éste se encargue de manipular el DOM.

2. **No se pueden usar ciertas funciones del navegador**

Muchas APIs y métodos que dependen del objeto window o de la interfaz gráfica no están disponibles, por ejemplo, no puedes llamar a `alert()`, `prompt()` o usar `document.createElement()` directamente dentro de un Worker. Sin embargo, sí puedes usar gran parte de las API web como `fetch()` y similares, siempre y cuando no requieran acceso directo al DOM.

3. **Sobrecarga de creación y comunicación**

Crear un Worker y comunicarse con él puede acarrear un pequeño coste adicional. Por ello, no es recomendable crear excesivos Workers para tareas muy pequeñas. Es preferible reservarlos para procesos que de verdad lo requieran, como transformaciones de grandes volúmenes de datos o cálculos matemáticos intensivos.

Además, cada Worker consume recursos (memoria y CPU). Un número muy elevado de Workers podría saturar el sistema.

4. **Estructura de archivos**

Los Workers deben ser definidos en archivos JavaScript separados e instanciarse con `new Worker('archivoWorker.js')`. No se puede “incrustar” el script de un Worker en una misma página HTML usando el mismo contexto, lo que obliga a separar el código y mantener una organización clara de ficheros.

5. **Sincronización y concurrencia**

Aunque los Workers proporcionan paralelismo, aún tenemos que pensar en la sincronización. Si varios Workers manipulan los mismos datos (por ejemplo, acceden a una misma base de datos externa), debemos coordinar esas operaciones de forma que no se produzcan inconsistencias.

6. **Utiliza técnicas de optimización**

Al transmitir datos muy grandes (arrays, objetos), considera usar `transferables` o técnicas de serialización eficientes para minimizar la sobrecarga de copia de datos.

Emplea algoritmos bien diseñados y evita recursividad extrema sin necesidad.

7. **Manejo adecuado de errores**

Manejar los eventos `onerror` en el Worker y en el hilo principal para detectar cualquier falla en la ejecución del script.

Recuerda que las excepciones que suceden dentro de un Worker no bloquearán el hilo principal, por lo que es importante llevar un registro de posibles errores.

10.6. EJEMPLOS

EJEMPLO 1: EVENT LOOP VS WEB WORKER

Código asíncrono tradicional (sin paralelismo real):

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8" />
  <title>Asincronía - Event Loop</title>
</head>
<body>
  <h2>Ejemplo de Asincronía con Event Loop</h2>
  <button id="btnIniciar">Iniciar Tareas</button>
  <div id="output"></div>

  <script>
    const boton = document.getElementById('btnIniciar');
    const salida = document.getElementById('output');

    boton.addEventListener('click', () => {
      salida.innerHTML = '';
      salida.innerHTML += '<p>Inicio de la operación (hilo principal)</p>';

      // Tarea asíncrona: se ejecutará después de 2 segundos
      setTimeout(() => {
        salida.innerHTML += '<p>Tarea asíncrona completada tras 2 segundos</p>';
      }, 2000);

      // Tarea sincrónica inmediata
```

```
    salida.innerHTML += '<p>Fin del click (hilo principal)</p>';
  });
 </script>
</body>
</html>
```

Código con Web Worker (paralelismo real):

index.hmtl

```
<!DOCTYPE html>
<html>
<head>
 <meta charset="UTF-8" />
 <title>Comunicación con Web Worker</title>
</head>
<body>
 <h2>Comunicación Básica con un Web Worker</h2>
 <button id="btnEnviar">Enviar Mensaje al Worker</button>
 <div id="respuesta"></div>
 <script>
  // 1. Crear la instancia de Worker
  const worker = new Worker('workerBasico.js');
  const boton = document.getElementById('btnEnviar');
  const respuestaDiv = document.getElementById('respuesta');
  // 2. Evento para enviar mensaje al Worker
  boton.addEventListener('click', () => {
   worker.postMessage('Hola, Worker!');
  });
  // 3. Escuchar respuestas del Worker
  worker.onmessage = (event) => {
   respuestaDiv.textContent = `Respuesta del Worker: ${event.data}`;
  };
 </script>
</body>
</html>
```

workerBasico.js

```
// Escuchar el evento onmessage para recibir mensajes del hilo principal
onmessage = function(event) {
  console.log('Mensaje recibido del hilo principal:', event.data);

  // Responder al hilo principal
  const respuesta = `Worker ha recibido tu mensaje: "${event.data}"`;
  postMessage(respuesta);
};
```

EJEMPLO 2: FIBONACCI CON WEB WORKER

Este ejemplo demuestra el proceso de realizar una operación costosa (cálculo recursivo de Fibonacci) en un Worker, evitando que la interfaz se bloquee.

fibonacci.html

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8" />
  <title>Fibonacci con Worker</title>
</head>
<body>
  <h2>Cálculo de Fibonacci usando Web Worker</h2>
  <label for="numeroFibo">Número de Fibonacci a calcular:</label>
  <input type="number" id="numeroFibo" value="40" />
  <button id="btnCalcular">Calcular</button>
  <div id="resultado"></div>
  <div id="estado"></div>
  <script>
    const input = document.getElementById('numeroFibo');
    const boton = document.getElementById('btnCalcular');
    const resultado = document.getElementById('resultado');
    const estado = document.getElementById('estado');
```

```
    // Creación del Worker
    const workerFibo = new Worker('workerFibonacci.js');
    // Escuchar mensajes del Worker (resultado)
    workerFibo.onmessage = (event) => {
      resultado.textContent = `Resultado: ${event.data}`;
      estado.textContent = 'Cálculo finalizado.';
    };
    // Manejo de posibles errores dentro del Worker
    workerFibo.onerror = (event) => {
      estado.textContent = `Error en Worker: ${event.message}`;
    };
    // Evento del botón para enviar petición de cálculo
    boton.addEventListener('click', () => {
      estado.textContent = 'Calculando...';
      resultado.textContent = '';
      const valor = parseInt(input.value, 10);
      workerFibo.postMessage(valor);
    });
  </script>
</body>
</html>
```

workerFibonacci.js

```
// Cálculo recursivo de Fibonacci
function fibonacci(n) {
  if (n <= 1) return n;
  return fibonacci(n - 1) + fibonacci(n - 2);
}
onmessage = function(event) {
  const numero = event.data;
  const resultado = fibonacci(numero);
  postMessage(resultado);
};
```

EJEMPLO 3: LIMITACIONES Y BUENAS PRÁCTICAS

A continuación se muestra que **no se puede acceder al DOM** desde el Worker y cómo pasar datos más grandes mediante mensajes. La idea es ilustrar buenas prácticas y limitaciones:

limitaciones.html

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8" />
  <title>Limitaciones y Buenas Prácticas con Workers</title>
</head>
<body>
  <h2>Limitaciones de Web Workers</h2>
  <button id="btnProcesar">Procesar Lista en Worker</button>
  <div id="output"></div>
  <script>
    const workerLim = new Worker('workerLimitaciones.js');
    const boton = document.getElementById('btnProcesar');
    const salida = document.getElementById('output');
    workerLim.onmessage = (event) => {
      // Recibimos un array procesado y lo mostramos
      const arrayProcesado = event.data;
      salida.textContent = `El Worker devolvió un array de longitud: ${arrayProcesado.length}`;
    };
    boton.addEventListener('click', () => {
      // Generamos un array grande para enviar al Worker
      const numeros = Array.from({ length: 100000 }, (_, i) => i);
      workerLim.postMessage(numeros);
    });
  </script>
</body>
</html>
```

workerLimitaciones.js

```
onmessage = function(event) {
  // No podemos manipular el DOM directamente; solo podemos procesar datos.
  const datos = event.data; // Este es el array recibido
  const resultado = datos.map(num => num * 2); // Ejemplo de procesamiento
  // Buenas prácticas: intentar no devolver datos excesivamente grandes
  // si no es necesario, y coordinar la comunicación de forma eficiente.
  postMessage(resultado);
};
```

Se observa que aquí **no** accedemos a `document`, `window` o `alert()` en el Worker porque no está permitido. Del mismo modo, se requiere una comunicación vía mensajes para retornar la información procesada.

EJEMPLO 4: CÁLCULO ESTADÍSTICOS SIN BLOQUEO DE INTERFAZ

Imaginemos que tenemos una aplicación que, mientras el usuario puede hacer clic en botones, se encarga de **calcular estadísticas** de un gran volumen de datos en segundo plano. Este ejemplo muestra cómo delegar esa carga a un Worker sin bloquear la UI.

estadisticas.html

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8" />
  <title>Ejemplo Conjunto 1 - Estadísticas con Worker</title>
  <style>
    #contenedorBotones {
      margin-bottom: 10px;
    }
    #estadisticas {
      margin-top: 10px;
```

```
    background-color: #f5f5f5;
    padding: 10px;
  }
 </style>
</head>
<body>
 <h2>Estadísticas con Web Worker</h2>
 <div id="contenedorBotones">
  <button id="btnGenerarDatos">Generar Datos y Calcular Estadísticas</button>
  <button id="btnOtraAccion">Otra Acción (UI disponible)</button>
 </div>
 <div id="estado">En espera...</div>
 <div id="estadisticas"></div>

 <script>
  const btnGenerar = document.getElementById('btnGenerarDatos');
  const btnOtraAccion = document.getElementById('btnOtraAccion');
  const estado = document.getElementById('estado');
  const estadisticasDiv = document.getElementById('estadisticas');

  // Creamos un Worker para el cálculo de estadísticas
  const workerStats = new Worker('workerEstadisticas.js');

  // Cuando el Worker termine de procesar, actualizamos la interfaz
  workerStats.onmessage = (event) => {
   const { media, min, max } = event.data;
   estado.textContent = 'Cálculo completado.';
   estadisticasDiv.innerHTML = `
    <p>Media de la muestra: ${media.toFixed(2)}</p>
    <p>Valor mínimo: ${min}</p>
    <p>Valor máximo: ${max}</p>
   `;
  };

  workerStats.onerror = (event) => {
   estado.textContent = `Error en Worker: ${event.message}`;
```

```
    };

    // Botón para generar datos y pedirle al Worker que calcule
    btnGenerar.addEventListener('click', () => {
      estado.textContent = 'Generando datos y calculando...';
      estadisticasDiv.innerHTML = '';

      // Generamos un array con 1 millón de números aleatorios
      const datos = Array.from({ length: 1000000 }, () => Math.random() * 1000);

      // Enviamos el array al Worker
      workerStats.postMessage(datos);
    });

    // Botón para realizar otra acción "simulada" en la UI
    btnOtraAccion.addEventListener('click', () => {
      alert('La interfaz sigue respondiendo mientras se calculan estadísticas!');
    });
  </script>
</body>
</html>
```

workerEstadisticas.js

```
onmessage = function(event) {
  const datos = event.data; // Array numérico
  // Calculamos la media, el valor mínimo y el valor máximo
  // (Aquí podríamos realizar otros cálculos más complejos)
  let suma = 0;
  let min = Number.MAX_VALUE;
  let max = -Number.MAX_VALUE;

  for (let i = 0; i < datos.length; i++) {
    const valor = datos[i];
    suma += valor;
    if (valor < min) min = valor;
```

```
    if (valor > max) max = valor;
  }

  const media = suma / datos.length;
  // Enviamos los resultados al hilo principal
  postMessage({ media, min, max });
};
```

- La interfaz no se bloquea mientras se generan y calculan estadísticas de un array con un millón de números.
- El usuario puede hacer clic en “Otra Acción” para ver un `alert()`, demostrando que la UI permanece responsiva.

EJEMPLO 5: PROCESAMIENTO DE IMAGENES Y ASINCRONISMO

En este ejemplo, vamos a simular la carga de una imagen (o un archivo binario) y su posterior procesamiento con un Worker, mientras también realizamos una petición asíncrona con `fetch()` desde el hilo principal. Este ejercicio muestra cómo conviven la asincronía del Event Loop (peticiones de red) con la paralelización que ofrece el Worker.

procesarImagen.html

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8" />
  <title>Ejemplo Conjunto 2 - Procesamiento de Imagen</title>
  <style>
    #imagenPreview {
      max-width: 300px;
      margin: 10px 0;
    }
    #log {
      background: #f0f0f0;
```

```
    padding: 10px;
    margin-top: 10px;
  }
 </style>
</head>
<body>
 <h2>Procesamiento de Imagen con Web Worker</h2>
 <input type="file" id="inputFile" accept="image/*" />
 <button id="btnProcesar">Procesar Imagen</button>

 <div>
  <img id="imagenPreview" src="" alt="Vista previa" />
 </div>

 <div id="log"></div>

 <script>
  const inputFile = document.getElementById('inputFile');
  const btnProcesar = document.getElementById('btnProcesar');
  const imagenPreview = document.getElementById('imagenPreview');
  const logDiv = document.getElementById('log');
  let imagenSeleccionada = null;
  const workerImg = new Worker('workerImagen.js');
  // Mostrar la imagen seleccionada en pantalla
  inputFile.addEventListener('change', (event) => {
   const file = event.target.files[0];
   if (!file) return;
   const reader = new FileReader();
   reader.onload = function(e) {
    imagenPreview.src = e.target.result;
    imagenSeleccionada = e.target.result; // Base64 de la imagen
   };
   reader.readAsDataURL(file);
  });

  // Procesar imagen en el Worker
```

```
    btnProcesar.addEventListener('click', () => {
      if (!imagenSeleccionada) {
        alert('Selecciona primero una imagen');
        return;
      }

      log('Enviando imagen al Worker para procesar...');
      // Enviamos la imagen codificada en Base64 al Worker
      workerImg.postMessage(imagenSeleccionada);
    });
    // Respuesta del Worker
    workerImg.onmessage = (event) => {
      // Recibimos la imagen procesada o un mensaje de estado
      if (event.data.tipo === 'log') {
        log(event.data.mensaje);
      } else if (event.data.tipo === 'resultado') {
        log('Imagen procesada con éxito. Puedes visualizar el resultado en consola o aplicarlo a un canvas.');
      }
    };
    // Ejemplo de asincronía adicional: una petición fetch que ocurre en el hilo principal
    // (Se ejecuta en paralelo a las operaciones del Worker)
    fetch('https://jsonplaceholder.typicode.com/posts/1')
      .then(response => response.json())
      .then(data => {
        log(`Petición fetch completada: Título del post: "${data.title}"`);
      })
      .catch(error => {
        log(`Error en fetch: ${error}`);
      });
    // Función para registrar mensajes en pantalla
    function log(mensaje) {
      logDiv.innerHTML += `<p>${mensaje}</p>`;
    }
  </script>
</body>
</html>
```

workerImagen.js

```
// Simulamos un procesamiento de imagen (por ejemplo, aplicando un filtro simple)
onmessage = function(event) {
  const imagenBase64 = event.data;
  // En un caso real, podríamos convertir la imagen a un ArrayBuffer,
  // aplicar filtros pixel a pixel y volver a convertir a Base64, etc.
  // Por simplicidad, simulamos un tiempo de procesamiento y enviamos logs intermedios
  postMessage({ tipo: 'log', mensaje: 'Iniciando procesamiento de imagen...' });

  // Simulamos un retardo (por ejemplo, 2 segundos)
  setTimeout(() => {
    postMessage({ tipo: 'log', mensaje: 'Procesamiento intermedio...' });
  }, 1000);
  setTimeout(() => {
    // Al finalizar, devolvemos el resultado (aquí, por ejemplo, podríamos devolver la imagen procesada)
    postMessage({ tipo: 'resultado', data: imagenBase64 });
  }, 3000);
};
```

- Este ejemplo combina la lectura de archivos (`FileReader`) y la transformación simulada en un Worker, junto con una petición `fetch` en el hilo principal.
- Mientras el Worker está ocupado, la petición `fetch` se realiza de forma asíncrona sin bloqueo.

10.7. CONCLUSIÓN

Los Web Workers representan uno de los avances más significativos para el desarrollo web moderno al permitir que JavaScript ejecute tareas intensivas de CPU sin bloquear la experiencia de usuario. Gracias a ellos, ahora es posible aprovechar múltiples hilos de ejecución y hacer un uso más eficaz de los recursos del sistema. No obstante, su uso no está exento de limitaciones, principalmente la imposibilidad de manipular el DOM directamente desde un Worker y la necesidad de comunicar información mediante mensajes.

En todo caso, cuando el volumen de datos a procesar sea alto, es buena idea recurri a los web workers. El diseño cuidadoso del intercambio de mensajes y la definición de responsabilidades claras entre el hilo principal y los Workers son claves para crear aplicaciones web avanzadas, de gran rendimiento y libres de bloqueos. A la hora de mostrar o actualizar datos en la interfaz, confía la responsabilidad al hilo principal. Delega a los Workers aquellas tareas que exijan una gran cantidad de recursos, como cálculos matemáticos complejos, procesado de ficheros o la preparación de grandes volúmenes de datos. De este modo, tu aplicación se beneficiará de una interacción ágil y una experiencia de usuario mucho más fluida.