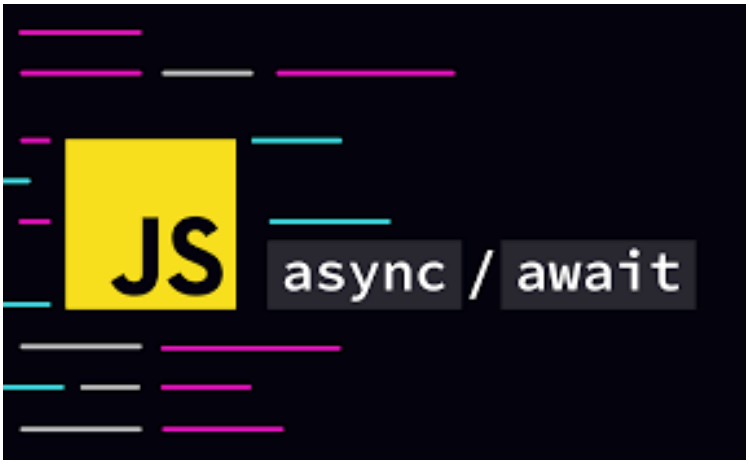
Javascript Asíncrono
y
Web Workers



Contenido

1. Código síncrono vs código asíncrono

JavaScript es un lenguaje de programación de un solo subproceso, lo que significa que solo puede suceder una cosa a la vez. Si bien un solo hilo simplifica la escritura y el razonamiento sobre el código, esto también tiene algunos inconvenientes.

Imagina que hacemos una tarea de larga duración como obtener un recurso a través de la red. Ahora bloqueamos el navegador hasta que se descargue el recurso. Esto puede generar una mala experiencia para el usuario y puede provocar que el usuario abandone nuestra página.

Cuando ejecutamos código de forma **síncrona**, esperamos a que finalice antes de pasar a la siguiente tarea. **No puede suceder nada más mientras se procesa cada operación: el procesamiento está en pausa**.

Ahí es donde entra en juego el JavaScript **asíncrono**. Usando JavaScript asíncrono, podemos realizar tareas de larga duración sin bloquear el hilo principal. Cuando ejecutamos algo de forma asíncrona, podemos pasar a otra tarea antes de que finalice.

El bucle de eventos es el secreto detrás de la programación asíncrona de JavaScript. JavaScript ejecuta todas las operaciones en un solo subproceso, pero el uso de algunas estructuras de datos inteligentes nos da la ilusión de subprocesos múltiples.

Así, por ejemplo, cuando buscamos una imagen de un servidor, no podemos devolver el resultado inmediatamente. Eso significa que lo siguiente no funcionaría:

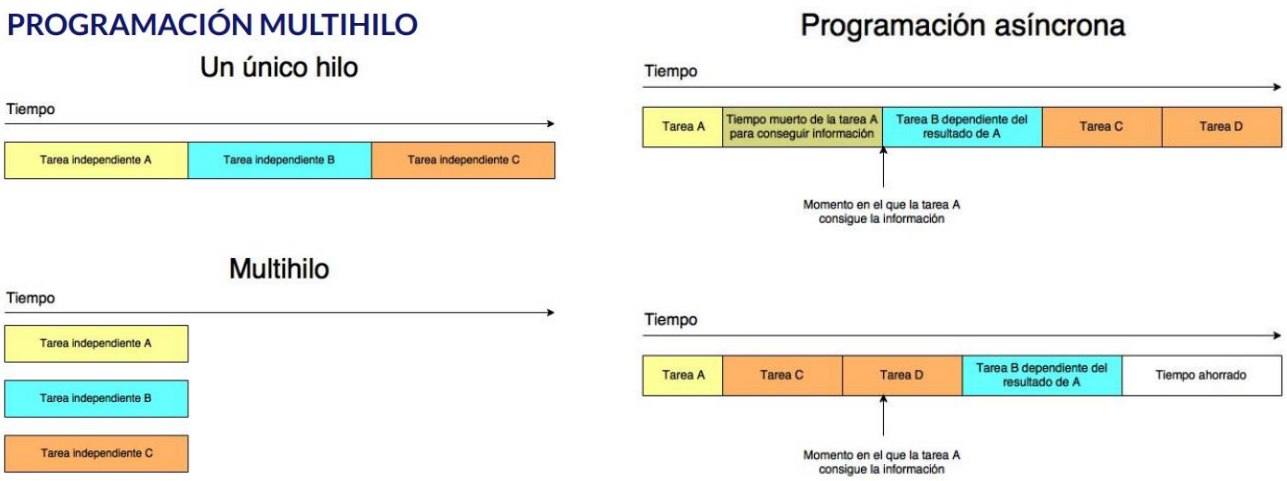
```
let respuesta = buscarEnServer('images/miImagen.png'); // buscar es asíncrono
let blob = respuesta.blob();
```

Eso es porque no sabemos cuánto tarda la imagen en descargarse, por lo que cuando ejecutamos la segunda línea, arroja un error porque la respuesta aún no está disponible. En cambio, debemos esperar hasta que regrese la respuesta antes de usarlo.

2. JavaScript asíncrono

Javascript fue diseñado para ser ejecutado en navegadores, trabajar con peticiones sobre la red y procesar las interacciones de usuario, al tiempo que se mantiene una interfaz fluida. Ser bloqueante o síncrono no ayudaría a conseguir estos objetivos, es por ello que Javascript ha evolucionado intencionadamente pensando en operaciones de tipo *I/O*. Por esta razón:

JAVASCRIPT UTILIZA UN MODELO **ASÍNCRONO Y NO BLOQUEANTE**, CON UN ***LOOP*** **DE EVENTOS IMPLEMENTADO CON UN ÚNICO *THREAD (HILO)*** PARA SUS INTERFACES DE ENTRADA/SALIDA.



Programación Multihilo vs Programación asíncrona

Gracias a esta solución, Javascript es altamente concurrente a pesar de emplear un único *thread*. Para ejecutar código asíncrono en JavaScript usaremos **eventos, promesas, funciones asíncronas, Web Workers, módulos y conexiones asíncronas con el servidor**. Éste último caso (conexiones asíncronas con el servidor) se verá en otro tema.

2.1. Promesas

Como acabamos de explicar, JavaScript tiene un único hilo de ejecución, de forma que hasta que no ejecuta una instrucción, no pasa a la siguiente. Eso puede llegar a producir retrasos indeseables, e incluso pérdida de control por parte del usuario.

Una promesa es un objeto que representa **el resultado de una operación asíncrona**. Este resultado podría estar disponible **ahora** o en el **futuro**. A continuación, se muestra un ejemplo sencillo de cómo se declararía y ejecutaría una promesa:

```
let miPromesa = new Promise (function (resolve, reject) {
    //Lógica de la función…
    [if (<condición>)]
       resolve ([valor]);
    [if(<error>]
      reject([valor]);
});

miPromesa.then(function (valorResolve) {
    console.log("Obtenido: " + valorResolve); // Ejecuta resolve.
```

```
}, function (error) {
    console.log("Error: " + error.message); // Ejecuta reject
};
```

Las promesas se crean con el operador ***new***. Una promesa ejecuta una función anónima a la que se le pasan dos parámetros, que son, a su vez, funciones (también llamados *callbacks*). El primero de esos parámetros se ejecutará cuando la promesa quiera devolver un valor válido (suele llamarse ***resolve***). El segundo se ejecutará cuando queramos que la promesa devuelva un error (ese parámetro de tipo function suele llamarse ***reject***).

Tanto **resolve** como **reject** solo pueden devolver un valor, eso sí, ese valor puede ser un objeto, un array o cualquier tipo de dato.

Habrá ocasiones en las que no necesitemos devolver ningún valor, solamente saber si la promesa ha acabado. En estos casos no será necesario utilizar **resolve** o **reject**. En otras ocasiones, puede que solo nos interese saber si la promesa ha acabado correctamente y obtener el valor. En esos casos solo usaremos ***resolve***.

Para recoger los valores, tanto válidos o erróneos, o saber cuándo ha acabado la promesa, utilizamos los dos parámetros de **then()** que son dos funciones anónimas: la primera se ejecuta cuando la promesa se ha ejecutado correctamente y la segunda cuando termina con errores. Así, otra forma de ejecutar las promesas puede ser la siguiente:

```
let miPromesa = new Promise (function (resolve, reject) {
    console.log ("Promesa pendiente");
    resolve ();
}).then(function () {
    console.log ("Promesa resuelta");
}, function (error) {
    console.log("Error: " + error.message);
});

console.log ("Hola mundo!");
```

La salida del código anterior será:

```
Promesa pendiente
Hola mundo!
Promesa resuelta
```

Lo que ha pasado es lo siguiente: lo primero que se ha lanzado es la función que se le pasa a la promesa, y pinta “Promesa pendiente”. Luego se devuelve el control a la línea de ejecución y pasa a la siguiente instrucción y escribe en la consola “Hola Mundo!”, y acto seguido se resuelve la promesa, en nuestro caso

cuando invocamos a resolve(), y entonces se lanza lo que hay en el then() y escribe "*Promesa resuelta*" en la consola.

Cuando se rechaza una promesa se puede ejecutar de dos formas: como la segunda función de un *then()* o como un *catch()*. El siguiente código sería equivalente:

```
miPromesa.then(function() {
  console.log("Promesa resuelta");
}, function(error){
  console.log("Error: " + error.message);
});

//-----------------------------------------------------

miPromesa.then(function() {
  console.log("Promesa resuelta");
})
.catch(function(error){
  console.log("Error: " + error.message);
})
.finally(function(){
  console.log("La promesa ya ha acabado");
});
```

Podemos simplificar la sintaxis de las promesas con ayuda de la función **arrow** (=>):

```
const myPromise = new Promise ((resolve, reject) => {
      // Código de la promesa…
});

myPromise.then (resolveValue => {
}
. catch (errorValue => {
});
```

A continuación, se muestra otro ejemplo más de uso de promesas. En él provocamos un retraso para provocar la asíncronía:

```
var myPromise = new Promise(function(resolve, reject) {
/* código de la función ejecutora */
 setTimeout(function() {
    resolve('Hola');
  }, 3000);
});

myPromise.then(function(value) {
  console.log(value); // Devolverá “Hola”
});
console.log(‘Otro código…’);
```

2.2. Promise.all, Promise.any, Promise.race y Promise.allSettled

Se pueden tener varias promesas, y querer que se lance algo cuando se cumplan todas o que se rechace con que una falle. Para ello usamos el método **Promise.all()**:

```
const p1 = Promise.resolve(1);
const p2 = Promise.reject(“Error”);
const p3 = Promise.resolve(3);
```

Ejecutamos directamente el método estático resolve o reject para simular el resultado de las promesas.

```
Promise.all([p1, p2, p3]).then(function () {
  console.log("Todas mis promesas resueltas");
}).catch(function(error){
  console.log("Error: " + error.message);
});
```

Además, tenemos **Promise.any()**. Su sintaxis es igual que *Promise.all*, acepta un array de promesas, pero en este caso, se ejecutará el then() si hay alguna promesa, **con una bastaría**, que se resuelva de forma correcta. Aquí tienes un ejemplo:

```
const promises = [
  fetch ('/from-external-api'),
  fetch ('/from-memory'),
```

fetch realiza peticiones al servidor y devuelve una promesa, se verá más adelante.

```
  fetch ('/from-new-api'),
]
try {
  // espera a la primera respuesta correcta que termine
  const first = await Promise.any(promises)
  console.log(first)
} catch (error) {
  // ¡Todas las promesas han fallado!
  console.assert(error instanceof AggregateError)
  console.log(error.errors)
}
```

NOTA: AggregateError

Como has podido ver en el ejemplo anterior, ahora cuando la promesa falla, se devuelve una instancia de *AggregateError*. Este error es una instancia del objeto ***Error*** y tiene una propiedad llamada *errors* que contiene una lista de errores para cada promesa que falló.

De igual forma tenemos el método **Promise.race()**. Su sintaxis es igual que *Promise.all*, acepta un array de promesas, pero en este caso, la primera promesa que termine provocará que se ejecute el then() o el catch() según dicha promesa haya terminado con *resolve()* o *reject()* respectivamente.

Por último, tenemos disponible el método **Promise.allSettled**. El método **Promise.allSettled** acepta una serie de promesas y solo se resuelve cuando todas están finalizadas, ya sea resueltas o rechazadas. Esto trae ***"Simplemente ejecute todas las promesas, no me importan los resultados"*** de forma nativa en JavaScript.

```
const myPromiseArray = [
      Promise.resolve(100),
      Promise.reject(null),
      Promise.reject(new Error('Oh no!'))
]
Promise.allSettled(myPromiseArray).then(results => {
      console.log('All Promises are finished', results)
})
```

Aquí tenéis un resumen de los métodos vistos:

Método	Descripción	Añadida en...
Promise.allSettled	Espera a todas las promesas se resuelvan o no	ES2020
Promise.all	Se para cuando una promesa es rechazada	ES2015
Promise.race	Se para cuando una promesa es rechaza o resuelta	ES2015
Promise.any	Se para cuando una promesa es resuelta	ES2021

2.3. Async y await

Básicamente es lo mismo que las promesas, para introducir asincronía, pero con una sintaxis ligeramente distinta y más sencilla.

Comienza con preceder una función de la palabra ***async*** para convertirla en una ***función asíncrona***. **Al invocar la función ahora devolverá una promesa.**

```
async function helloWorld(){
    return "hello world"; // Equivale a return Promise.resolve("hello world");
}
let hello = helloWorld()
console.log(hello)

// Salida: una promesa:
// Promise { 'hello world' }
```

async se puede ejecutar por sí solo, pero obtiene real eficiencia cuando se usa conjuntamente con **await**.

Con ***await*** especificamos la parte del código que debe esperar a resolver una promesa sin la que no puede continuar ejecutándose, es decir, su función es **pausar la ejecución de la función** hasta que la promesa se resuelva o se rechace. Para ello, se precede de la palabra reservada ***await***, esto nos indicará donde se genera la pausa del flujo del programa, **mientras se puede pasar a ejecutar otro código distinto**. Una vez resuelta, se sigue con la ejecución de la función asíncrona.

```
function resolverDespuesDe2Segundos() {
    return new Promise(resolve => {
      setTimeout(() => {
        resolve("resuelta!");
      }, 2000);
    });
  }

  async function asyncCall() {
    console.log("Llamando…");
    var result = await resolverDespuesDe2Segundos();
    console.log(result);  // Salida esperada: "resuelta!"
  }

  asyncCall();
  console.log ("El código sigue ejecutándose...");
```

Otro ejemplo:

```
const count = 100;

function promiseSqrt(value) {
    return new Promise(function (resolve, reject) {
        console.log('START execution with value = '+ value);
        setTimeout(function() {
            resolve({ value: value, result: value * value });
        }, 1500);
    });
}

async function run() {
    for (let n = 0; n <= 9; n++) {
        var obj = await promiseSqrt(n);
        console.log('END execution with value = '+ obj.value+ ' and result = '+
obj.result);
    }
}

let myPromise = run();
console.log(myPromise);

// Probamos que el código sigue ejecutándose mientras la promesa está pendiente:
for (let i = 0; i < count; i++) {
    console.log("Value of i = " + i);
}
```

Un ejemplo más:

```
async function sayHello() {

    /* PROBAR ESTE EJEMPLO Y EJECUTAR PASO A PASO AÑANDIENDO Y QUITANDO await a la siguiente
línea*/

    await setTimeout(function() {

       console.log("executing...")

    }, 3000)

    for (let i = 0; i < 10; i++)

        console.log(i+" - ")

    return "Hello";

}
```

```
let afn = sayHello(); // Se devuelve una promesa


console.log("Valor devuelto --> " + afn)


afn.then(function(val) {
    console.log("Valor devuelto por la Promesa: " + val)
}).catch(function() {
    console.log("ERROR!!")
});
```

Await a nivel superior: Podemos usar ***await*** sin necesidad de utilizarlo dentro de una función asíncrona con async. Ejemplo:

```
const datos = await fetch("https://conseguir-usuarios.json");
const usuarios = await datos.json();
```

3. Web Workers

Al ejecutar un script en una página HTML, la página deja de responder hasta que finaliza este script.

Un ***Web Worker*** es un script JavaScript que se ejecuta en segundo plano, independientemente de otros scripts, sin afectar el rendimiento de la página. Puede continuar haciendo lo que quiera: hacer clic, seleccionar cosas, etc., mientras el Web Worker se ejecuta en segundo plano.

Esta API funciona de forma **asíncrona**, disponiendo de dos métodos de uso. Nosotros solo nos centraremos en el que tiene compatibilidad con todos los navegadores modernos.

Antes de usar Web Workers, podemos chequear si el navegador lo suporta de la siguiente forma:

```
if (typeof(Worker) !== "undefined") {
  // Yes! Web worker support!
  // Some code.....
} else {
  // Sorry! No Web Worker support..
}
```

Para usar los Web Workers seguiremos los siguientes pasos:

1. Crearemos un fichero .js con el código que se ejecutará en segundo plano.
2. En este código que se ejecuta en segundo plano, podremos usar el método ***postMessage(<lo que sea>)***, el cual devuelve el valor ***<lo que sea>*** al HTML principal.
3. Ahora que tenemos creado el Web Worker, tenemos que llamarlo desde la página HTML o, desde el fichero JavaScript principal que enlaza la página HTML. Para ejecutar el Web Worker usaremos:

```
var w = new Worker("demo_workers.js");
```

4. Una vez hecho esto, podemos recibir y mandar mensajes entre el JavaScript principal y el Web Worker:

```
w.onmessage = function(event){
  document.getElementById("result").innerHTML = event.data;
};
```

Cuando el web worker publica un mensaje, se ejecuta el código dentro del detector de eventos. Los datos del web worker se almacenan en ***event.data***. Otra forma de crear el evento que enviará los datos al Web Worker sería:

```
worker.addEventListener('message', function(e) {
     alert('Respuesta del Worker: ' + e.data);
}, false);
```

5. Desde el Web Worker mandamos datos al JavaScript principal usando el método ***postMessage(<lo que sea>)***.
6. Por último, finalizamos la ejecución del Web Worker con el método ***terminate()***:

Aquí tienes un ejemplo completo:

Fichero web worker (ej: webWorker.js):

```
var i = 0;

function timedCount() {
  i = i + 1;
  postMessage(i);

// Ejecutamos el método timedCount cada 500 milisegundos:
  setTimeout("timedCount()",500);
}

timedCount();
```

Fichero JavaScript principal:

```
var w;

function startWorker() {
```

```
 // Verificamos si el navegador soporta Web Worker:
if (typeof(Worker) !== "undefined") {
     // Creamos el Web Worker si no está creado:

    if (typeof(w) == "undefined")
        w = new Worker("demo_workers.js");

     w.onmessage = function(event) {
        document.getElementById("result").innerHTML = event.data;
     };
  }

  else
     document.getElementById("result").innerHTML = "Sorry! No Web Worker support.";

}

function stopWorker() {
  w.terminate();
  w = undefined;
}
```

De igual forma, podemos comunicarnos desde el fichero JavaScript principal con el WebWorker. Para enviar datos al WebWorker usaremos la misma sintaxis ya vista:

1. Usaremos el método *postMessage()* desde el fichero JavaScript para enviar datos al WebWorker.
2. En el WebWorker usaremos el método *onmessage()* para crear el evento que espera el mensaje desde el fichero JavaScript. **Para hacer esto, deberemos usar *self.onmessage()* o *this.onmessage()* en el WebWorker.**