Programación con JavaScript II

Asincronía

Introducción

Algunos programas informáticos, como las simulaciones científicas y los modelos de aprendizaje automático, están vinculados a la computación: se ejecutan continuamente, sin pausa, hasta que calculan su resultado. Sin embargo, la mayoría de los programas informáticos del mundo real son significativamente asincrónicos, esto implica que a menudo tienen que dejar de computar mientras esperan que lleguen los datos o que ocurra algún evento.

Los programas de JavaScript en un navegador web suelen estar controlados por eventos, lo que significa que esperan a que el usuario haga clic o toque antes de hacer algo. Por su parte, los servidores basados en JavaScript suelen esperar a que lleguen las solicitudes de los clientes a través de la red antes de hacer algo.

Este tipo de programación asíncrona es común en JavaScript, y este tema documenta tres características importantes del lenguaje que ayudan a que sea más fácil trabajar con código asíncrono. Las promesas, nuevas en ES6, son objetos que representan el resultado aún no disponible de una operación asincrónica. Las palabras clave *async* y await se introdujeron en ES2017 y proporcionan una nueva sintaxis que simplifica la programación asincrónica al permitir estructurar el código basado en promesas como si fuera sincrónico. Finalmente, los iteradores asincrónicos y el bucle *for/await* se introdujeron en ES2018 y permiten trabajar con flujos de eventos asincrónicos usando bucles simples que parecen sincrónicos.

Explicación

Cuando se comienza en el desarrollo de aplicaciones, normalmente se realizan tareas de forma síncrona, esto es, que se programan las tareas para que se ejecuten una después de la otra o, dicho de otra manera, de manera secuencial. No obstante, tarde o temprano será necesario realizar operaciones en las que se tiene que esperar a que ocurra determinado evento que no depende precisamente del desarrollador. La desventaja de realizar este tipo de operaciones es que, como se debe esperar a que el o los procesos terminen, se bloquea cualquier otro proceso.

En un ambiente síncrono, la solución a este tipo de problemas es crear un nuevo hilo *(thread)*. Podemos definir a este thread como un programa que está ejecutándose y que en cualquier momento se puede intercalar con otros programas que estén corriendo en el sistema operativo. Lo anterior es posible gracias a que la mayoría de los equipos de cómputo modernos cuentan con más de un procesador. Dicho en otras palabras, podemos iniciar las dos tareas en diferente procesador y, una vez que terminan sus procesos, se sincronizan para combinar sus resultados.

Con JavaScript no se puede realizar esto, ya que, por definición, es un lenguaje de programación de un solo hilo (*single thread*). El motor de JavaScript solo procesa una sentencia a la vez en un thread único, de tal forma que, si se está realizando un proceso como el ejemplo anterior, el thread principal se bloquea hasta que la operación termina. Para el navegador, esto significa que la aplicación web se bloquea hasta que la operación finaliza.

Event loop

El event loop que utiliza JavaScript se puede representar con el siguiente gráfico:

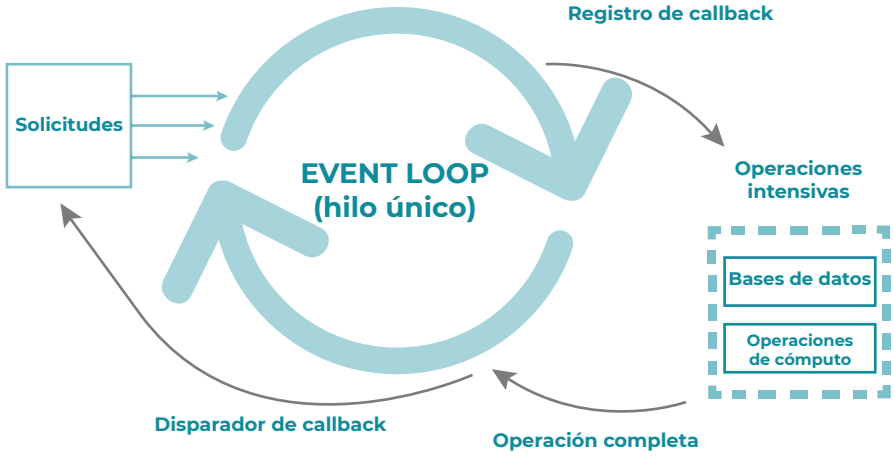

Figura 1. Event loop.

Explicación

Antes que nada, es preciso explicar que el *runtime* de JavaScript maneja dos elementos principalmente:

- *Memory heap:* es una manera desorganizada de guardar información en la memoria.
- *Call stack:* es la manera de ordenar las funciones que son invocadas.

Para efectos del event loop, solo se abordará call stack. Esta estructura simple es una pila en la cual se ingresa una función que está a punto de ser ejecutada, si esta función llama, a su vez, a otra función, esta es agregada encima de la anterior, de tal manera que se utiliza el método LIFO (*Last Input, First Output*). Veamos el siguiente ejemplo:

```
function multiply (x, y) {
return x * y;
}
function cuadrado (x) {
var r = multiply(x, x);
console.log(r);
}
cuadrado(5);
```

Figura 2. Función en JavaScript.

Los estados del call stack serían los siguientes:

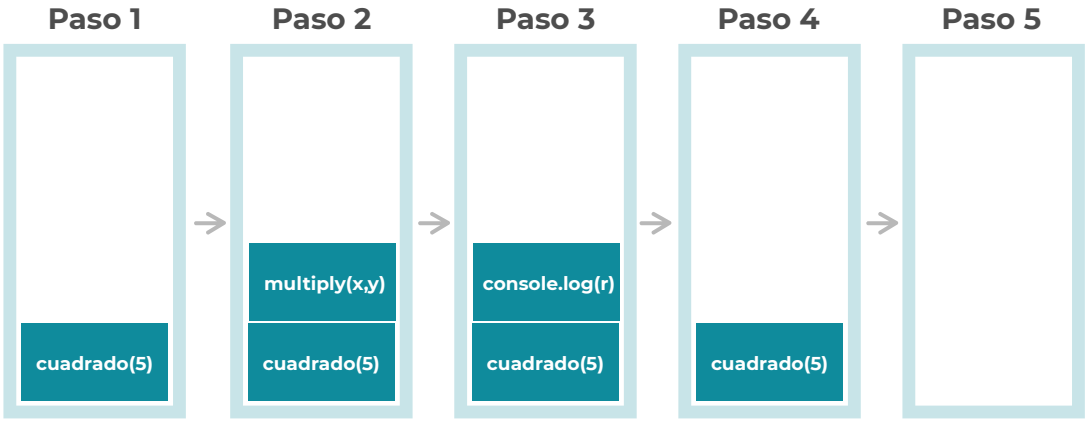

Figura 3. Estado de call stack.

Explicación

Y si se tiene una función como la siguiente:

```
function foo() {
foo();
}
foo();
```

Figura 4. Función foo.

Los estados del call stack serían los siguientes:

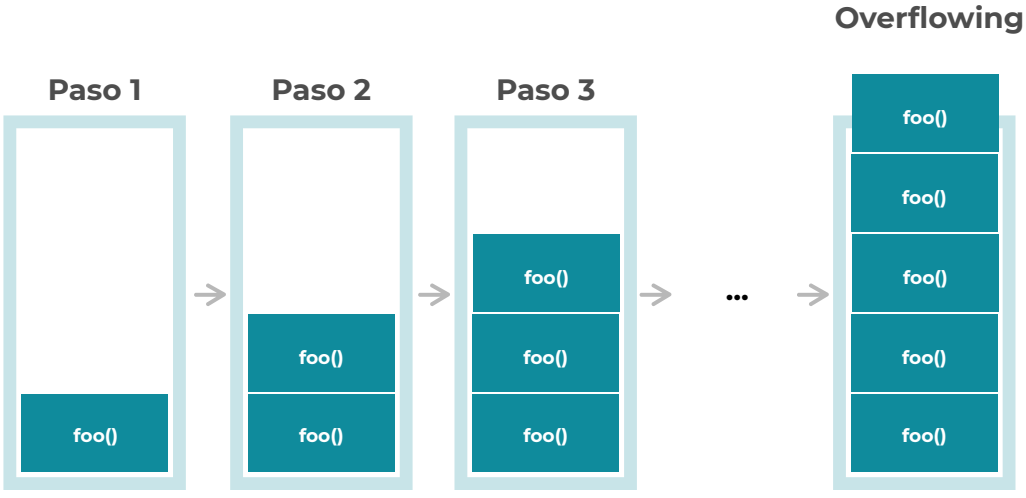

Figura 5. Estado de call stack.

Esto provocaría que, en algún momento, como se ve en la figura anterior, la cantidad de funciones llamadas exceda el tamaño de la pila, por lo que el navegador mostraría un error semejante al siguiente:

```
▶Uncaught RangeError: Maximum call stack size exceeded
```

Figura 6. Mensaje de error del navegador.

Pero si se llama a un *timeout* o se hace una solicitud con AJAX, en JavaScript, al ser de un solo hilo, por tanto, solo existe un call stack (se puede ejecutar un solo proceso a la vez), el navegador se congelaría y no se podría hacer nada más hasta que el proceso termine de ejecutarse. Esto sucede con procesos **asíncronos bloqueantes**. Pero JavaScript puede ser **asíncrono no bloqueante** gracias al event loop.

Explicación

Algo interesante del motor de JavaScript es que no cuenta de manera nativa con elementos como *setTimeout*, elementos de manipulación del DOM o HTTP *request*. Todas estas se hacen mediante llamadas a web API provistas por el navegador.

En el siguiente gráfico se muestra una visión de cómo funciona JavaScript.

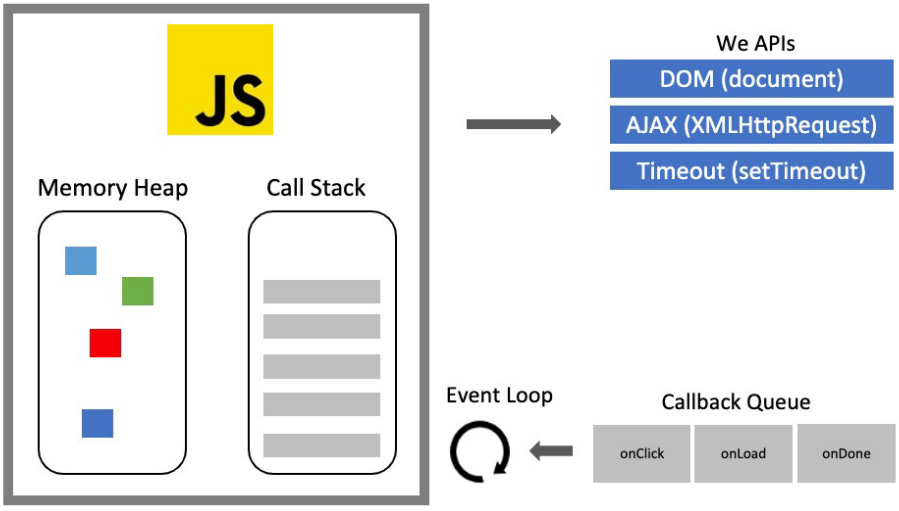

Figura 7. Funcionamiento de JavaScript.

Debido a que solamente hay un thread, es muy conveniente no desarrollar código bloqueante para que la UI no quede bloqueada, pero ¿cómo se hace eso?

Actualmente en JavaScript existen diferentes mecanismos mediante los cuales se pueden realizar operaciones con tiempos de respuesta largos sin que se bloquee el thread.

Callbacks

Según el enfoque de Kantor (s.f.), muchas funciones proporcionadas por el entorno de JavaScript permiten programar asíncronamente, es decir, son acciones que se inician ahora, pero que se terminan después. Para lograr esto, estas funciones deben tomar un argumento adicional, a lo que se le llama función de devolución de llamada (*callback*). Se puede tomar como ejemplo la función setTimeout, cuya funcionalidad es esperar una cantidad de milisegundos y pasado ese tiempo llama a la función. Veamos el siguiente ejemplo:

.

```
function cargaScript(origen) {
let codigo = document.createElement('script');
codigo.src = origen;
document.head.append(codigo);
```

Figura 8. Definición de función asíncrona.

Explicación

Ahora se llamaría así a la función:

```
cargaScript('/mi/script.js');
```

Figura 9. Llamada de función asíncrona.

El *script* se ejecutará asincrónicamente porque comienza a cargarse desde el momento de la llamada (ahora), pero se ejecuta hasta que la función haya terminado (más tarde). Lo más importante es que el código que se encuentre debajo de la llamada de la función no tendrá que esperar a que termine la carga del script. Ahora imagina que necesitas usar el nuevo script en cuanto se cargue, declarar nuevas funciones y que se ejecuten. Si se hace esto inmediatamente después de llamar a la función **cargaScript,** no funcionará:

```
cargaScript('/mi/script.js'); // tiene "function newFunction() {…}"
newFunction(); // no existe dicha función!
```

Figura 10. Llamada de función inexistente.

Esto se debe a que posiblemente el navegador no tuvo tiempo de cargar el script. Por el momento, la función recientemente creada **cargaScript** no da una manera de monitorear cuándo finaliza la carga. Para efectos prácticos, el script se carga y se ejecuta, sin más, pero a nosotros nos gustaría saber cuándo ocurre para poder utilizar las funciones y/o variables producto del script.

Para lograrlo, se agrega la función callback como segundo argumento de la función **cargaScript**, la cual deberá ejecutarse cuando se carga el script:

```
function cargaScript(origen, callback) {
        let codigo = document.createElement('script');
        codigo.src = origen;
        codigo.onload = () => callback(codigo);
        document.head.append(codigo);
```

Figura 11. Adición de función callback.

Explicación

Por lo tanto, si queremos llamar a las funciones, se deberá hacerlo desde callback:

```
cargaScript('/mi/script.js', function () {
        //callback se ejecuta luego que se carga el script
        newFunction(); // ahora funciona
...
});
```

Figura 12. Llamada de función callback.

Veamos el código en acción:

```
function cargaScript(origen, callback) {
        let codigo = document.createElement('script');
        codigo.src = origen;
        codigo.onload = () => callback(codigo);
        document.head.append(codigo);
}

// Uso
cargaScript('https://cdnjs.cloudflare.com/ajax/libs/lodash.js/3.2.0/lodash.js',
codigo => {
        alert(`Excelente, el script ${codigo.src} cargó correctamente`);
        alert(_); // función declarada en el script cargado
});
```

Figura 13. Ejemplo de callback.

¿Y qué pasaría si queremos un script más?

```
cargaScript('/mi/script.js'), function (codigo) {
      cargaScript('/mi/script2.js', function (codigo) {
            cargaScript('/mi/script3.js', function (codigo) {
                  // ...continúa después que se han cargado todos los scripts
            });
      });
}
```

Figura 14. Callback de callback.

Explicación

Hasta este momento se han tratado con condiciones ideales, pero ¿qué pasa si en algún momento ocurre un error? Para manejar estos eventos, se debe optimizar el código y que el callback pueda reaccionar antes:

```
codigo.onload = () => callback(codigo);
codigo.onerror = () => callback(new Error(`Error de carga de script con
${origen}`));
document.head.append(codigo);
...
cargaScript('/mi/script.js', function (error, codigo) {
        if (error) {
                // maneja el error
        } else {
                // script cargado satisfactoriamente
}
});
```

Figura 15. Manejo de errores con callbacks.

Las devoluciones de llamada son versátiles, pues no solo permiten controlar el orden en que se ejecutan las funciones y qué datos se pasan entre ellas, sino que también permiten pasar datos a diferentes funciones según las circunstancias. Por lo tanto, podrían tener diferentes acciones para ejecutar en la respuesta descargada, como processJSON (), displayText (), etcétera.

Haverbeke (2018) advierte que, en cierto modo, la asincronicidad es contagiosa. Además, hace énfasis en que cualquier función que llame a una función que se ejecute de forma asincrónica debe ser asincrónica en sí misma y que esta debe utilizar una devolución de llamada o un mecanismo similar para entregar su resultado. Se debe tener cuidado al llamar a una devolución de llamada debido a que es algo más complicado y propenso a errores que simplemente devolver un valor, por lo que la necesidad de estructurar grandes partes del programa de esa manera no es la mejor opción.

Se pueden identificar dos categorías principales de deficiencias en el uso de devoluciones de llamada para expresar la asincronía del programa y administrar la concurrencia: falta de secuencialidad y falta de confiabilidad. Ahora que se entienden los problemas de manera más íntima, es hora de prestar atención a los patrones que pueden abordarlos.

El problema que queremos abordar primero es la inversión del control, la confianza que se mantiene tan frágilmente y se pierde tan fácilmente.

Explicación

Promesas

En estos momentos se confía la continuación del programa a una función de devolución de llamada y se espera que el resultado que regrese se entregue a otra parte (potencialmente incluso un código externo), en otras palabras, solo queda cruzar los dedos para que haga lo correcto con la invocación de la devolución de llamada.

Pero ¿y si se pudiera deshacer esa inversión de control?, ¿qué pasa si en lugar de entregar la continuación del programa a otra parte, pudieras esperar que devuelva la capacidad de saber cuándo finaliza su tarea y luego el código pudiera decidir qué hacer a continuación?

A esto le vamos a llamar promesa. Según Flanagan (2020), una promesa es un objeto que representa el resultado de un cálculo asincrónico y que nos indica/advierte que ese resultado puede o no estar listo todavía. La API de promesa es intencionalmente vaga al respecto: no hay manera de obtener de forma sincrónica el valor de una promesa, solo se le puede pedir a promesa que llame a una función callback cuando el valor esté listo e informar a cualquier persona que esté interesada.

Entonces, en el nivel más simple, las promesas son solo una forma diferente de trabajar con devoluciones de llamada. Sin embargo, existen beneficios prácticos al usarlas. Un problema real con la programación asincrónica basada en devoluciones de llamada es que es común terminar con devoluciones de llamada dentro de las devoluciones de llamada, con líneas de código tan sangradas o indentadas que son difíciles de leer. Las promesas permiten que este tipo de devolución de llamada anidada se vuelva a expresar como una cadena de promesas más lineal que tiende a ser más fácil de leer y de razonar.

Otro problema con las devoluciones de llamada es que pueden dificultar el manejo de errores. Si una función asincrónica (o una devolución de llamada invocada de forma asincrónica) arroja una excepción, no hay forma de que esa excepción se propague de nuevo al iniciador de la operación asincrónica. Este es un hecho fundamental sobre la programación asincrónica: rompe el manejo de excepciones.

La alternativa para esto es rastrear y propagar meticulosamente los errores con argumentos de devolución de llamada y valores de retorno, pero esto es tedioso y difícil de hacer bien. Las promesas ayudan aquí al estandarizar una forma de manejar los errores y proporcionar una manera para que los errores se propaguen correctamente a través de una cadena de promesas.

Se debe tener en cuenta que las promesas representan los resultados futuros de cálculos asincrónicos únicos, sin embargo, no se pueden utilizar para representar cálculos asincrónicos repetidos.

Explicación

Analicemos la forma de ponerlo en práctica. Esta es una implementación de una función basada en promesa que suma dos números: a y b:

```
function sumAsync(a, b) {
    return new Promise(
        (resolve, reject) => { // (línea A)
            if (a === undefined || b === undefined) {
                reject(new Error('Debe ingresar dos
                parámetros'));
            } else {
                resolve(a + b);
            }
        });
```

Figura 16. Manejo de errores con promesas.

En este caso, **sumAsync()** inmediatamente invoca al constructor promesa. La implementación real de esa función reside en la devolución de llamada que se pasa a ese constructor (línea A). Esa devolución de llamada tiene dos funciones:

- ***Resolve:*** se utiliza para entregar un resultado (en caso de éxito). Esta función se asegura de que el valor que se le asigne sea envuelto en una promesa, por tanto, si ya es una promesa, simplemente es retornada; en otro caso, se obtiene una nueva promesa.
- ***Reject:*** se utiliza para entregar un error (en caso de falla).

Conforme a lo que explica Rauschmayer (2021), existen tres estados en los que puede estar una promesa: pendiente, cumplida o rechazada. Además, puede tener tres resultados: indefinido, valor (cuando se llama a **resolve**) o error (cuando se llama a **reject**).

No se puede acceder directamente a las propiedades **estado** y **resultado** de un objeto promesa, para ello, necesitamos utilizar los métodos: *.then, .catch* y *.finally*, a los que llamaremos consumidores.

```
let veinte = Promise.resolve(20);
veinte.then(valor => console.log(`Obtuve ${valor}`));
// → Obtuve 20
```

Figura 17. Manejo de errores con consumidores.

Se pueden agregar múltiples devoluciones de llamada a una sola promesa. Estas se llamarán incluso si se agregan después de que la promesa tenga el estatus de cumplida (finalizada).

Explicación

.then es el consumidor más importante. Su primer argumento es una función que cuando se resuelve la promesa, recibe el resultado, mientras que el segundo argumento es una función que cuando se rechaza la promesa, recibe error.

```
sumAsync(4)
        .then(result => { // éxito
                console.log(result);
        },
                error => { // error
                        console.log(error);
                }
        )
```

Figura 18. Uso de .then

En el caso de que los parámetros sean correctos, se regresa ***result,*** en caso contrario, regresa ***error.*** En el uso del consumidor .catch, solo van a interesar los errores, por lo cual se pueden usar expresiones como .catch(errorHandlingFunction). Se puede implementar de la siguiente manera:

```
sumAsync(4)
        .catch(
        error => { // error
                console.log(error);
        }
)
```

Figura 19. Uso de .catch

La llamada .catch(f) es similar a .then(null,f), considerando que f se ejecuta.

El consumidor finally es similar a .then(f,f) y es una buena implementación para hacer la limpieza. El siguiente ejemplo dará una mejor perspectiva de su uso:

```
sumAsync(4)
        .finally(()=>alert("Pomesa lista"))
        .then(result => alert(result))
        .catch(error => alert(error))
```

Figura 20. Uso de .finally

Como lo explica Kantor (s.f.), si hay una promesa pendiente, los tres manejadores que acabamos de analizar la esperan, pero se ejecutan inmediatamente si esa promesa ya se resolvió.

Explicación

Ahora se reescribe el código de la figura 13 usando promesas:

```
function cargaScript(origen) {
return new Promise(function (resolve, reject) {
let codigo = document.createElement('script');
codigo.src = origen;
codigo.onload = () => resolve(codigo);
codigo.onerror = () => reject(new Error(`Error de carga de script para $ {origen}`));
document.head.append(codigo);
});
}
// Uso
let promesa =
cargaScript('https://cdnjs.cloudflare.com/ajax/libs/lodash.js/3.2.0/lodash.js');
promesa.then(
codigo => alert(`Excelente, el script ${codigo.src} cargo correctamente`),
error => alert(`Error: ${error.message}`)
);
```

Figura 21. Uso de promesas.

Funciones asíncronas

Una manera más práctica de trabajar con promesas es utilizando *async/wait*. Para trabajar con las funciones async solo se requiere anteponer la palabra clave antes de la función:

```
async function f() {
return 1;
}
```

Figura 22. Async.

Esto mismo se puede expresar como una función flecha de la siguiente manera:

```
const f= async ()=> 1;
```

Figura 23. Función flecha.

Explicación

Cuando colocamos la palabra “async” antes de una función, significa que la función devolverá una promesa. Otros valores se devolverán y resolverán en una promesa de manera automática, por ejemplo, la siguiente función devuelve una promesa con resultado 1:

```
const f= async ()=> 1;
f().then(alert); // 1
```

Figura 24. Resultado 1.

Esto se podría realizar también de la siguiente manera, regresando explícitamente la promesa:

```
const f= async ()=> Promise.resolve(1);
f().then(alert); // 1
```

Figura 25. Regresando promesa.

Resumiendo, async hace que la función devuelva una promesa o regresa no promesas y las convierte en una.

Las funciones definidas con async o cualquier promesa puede utilizarse en conjunto con await. Lo que hace dicha palabra reservada es esperar a que la promesa responda y devuelva el resultado y permite continuar con la ejecución de otras operaciones durante ese tiempo. Por tanto, no cuesta ningún recurso al CPU durante la espera. Por ejemplo:

```
async function f() {
let promise = new Promise((resolve, reject) => {
setTimeout(() => resolve("¡Hola!"), 1000)
});
let result = await promise; // espera hasta que la promesa se resuelva (*)
alert(result); // "¡Hecho!"
}

f();
```

Figura 26. Pausar función.

En este ejemplo se pausa la ejecución de la función y se reanuda cuando la promesa responde, un segundo después.

Cierre

Como pudiste aprender en este tema, el paralelismo en los procesos no siempre puede resolver el problema de la concurrencia, por lo que es necesario recurrir a la programación asincrónica.

JavaScript puede trabajar con programación síncrona y asincrónica. También aprendiste que la forma de utilizar la asincronía en JavaScript es a través del uso de:

- **Callbacks:** función que se va a ejecutar cuando una operación asíncrona termina, como resultado de esta.
- **Promesa**: es el resultado de una operación asíncrona. Se configura con dos llamadas para informar el éxito o el fallo de esta.
- **Funciones asíncronas:** funciones para gestionar promesas de forma más sencilla.

Referencias bibliográficas

- Flanagan, D. (2020). *JavaScript: The Definitive Guide* (7a ed.). Estados Unidos: O'Reilly Media, Inc.
- Haverbeke, M. (2018). *ELOQUENT JAVASCRIPT* (3ª ed.). Estados Unidos: No Starch Press.
- Kantor, I. (s.f.). *The Modern JavaScript Tutorial.* Recuperado de https://javascript.info/
- Rauschmayer, A. (2021). *JavaScript For Impatient Programmers (ES2021 edition).*

Para saber más

Lecturas

- MDN. (s.f.). *JavaScript asíncrono.* Recuperado de https://developer.mozilla.org/es/docs/Learn/JavaScript/Asynchronous
- Lenguaje JS. (s.f.). *¿Qué es la asincronía?* Recuperado de https://lenguajejs.com/javascript/asincronia/que-es/

Videos

- Paradigma Digital. (2020, 17 de enero). *Curso de Programación asíncrona en JavaScript* [Archivo de video]. Recuperado de https://www.youtube.com/watch?v=L8ogb9XO1hg
- Carlos Azaustre – Aprende JavaScript. (2020, 23 junio). *¿Cómo funcionan las Promises y Async/Await en JavaScript? [2022]* [Archivo de video]. Recuperado de https://www.youtube.com/watch?v=rKK1q7nFt7M

Checkpoints

Asegúrate de:

- Comprender la diferencia entre programación síncrona y asíncrona.
- Comprender y utilizar correctamente las funciones callback.
- Identificar cuándo se debe utilizar una promesa.
- Comprender qué son las funciones asíncronas.

Requerimientos técnicos

- Computadora con acceso a Internet.
- Editor de texto.
- Permisos de administrador y/o Git instalado previamente.

Prework

Los siguientes enlaces son externos a la Universidad Tecmilenio, al acceder a ellos considera que debes apegarte a sus términos y condiciones:

- Descarga la herramienta Git del sitio oficial dependiendo de tu sistema operativo y crea una cuenta en GitHub. Si no sabes cómo hacerlo, ve a la página: https://docs.github.com/es/desktop o https://git-scm.com/book/es/v2
- Crea un directorio principal donde coloques todos tus repositorios y archivos.
- Crea una carpeta específica para tu primer proyecto en Git.
- Asegúrate de tener instalado un entorno de desarrollo como Visual Studio Code o un editor de código como Sublime Text. Cualquiera de los sugeridos está bien, incluso puedes utilizar uno en línea, por ejemplo, https://playcode.io/.
- Se requiere que tengas un navegador web instalado, de preferencia Google Chrome en su última versión.