

# JavaScript Asíncrono

Última actualización: Abr 15 2020



## Asincronía

La asincronía es uno de los pilares fundamentales de *Javascript*, ya que es un lenguaje de programación de un sólo subproceso o hilo (*single thread*), lo que significa que sólo puede ejecutar una cosa a la vez.

Si bien los idiomas de un sólo hilo simplifican la escritura de código porque no tiene que preocuparse por los problemas de concurrencia, esto también significa que no puede realizar operaciones largas como el acceso a la red sin bloquear el hilo principal.

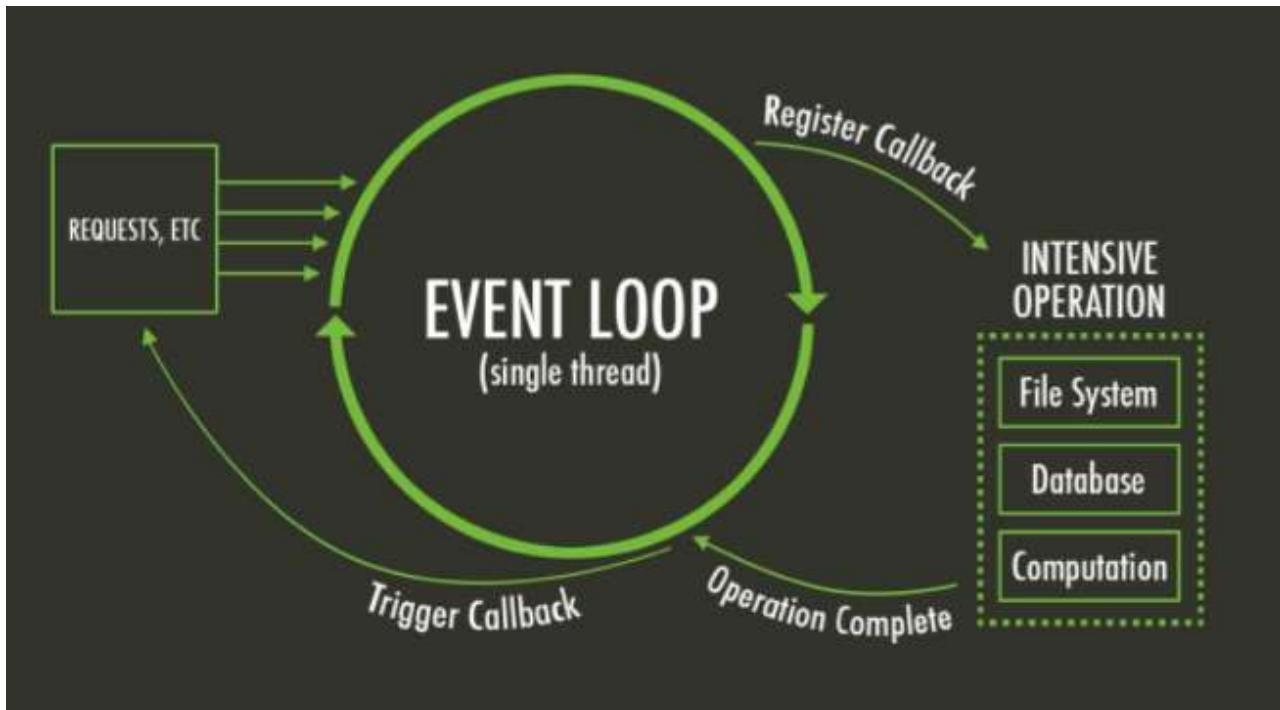
Imagina que solicitas datos de una *API*. Dependiendo de la situación, el servidor puede tardar un tiempo en procesar la solicitud mientras bloquea el hilo principal y hace que la página web no responda.

Ahí es donde entra en juego la asincronía que permite realizar largas solicitudes de red sin bloquear el hilo principal.

*JavaScript* fue diseñado para ser ejecutado en navegadores, trabajar con peticiones sobre la red y procesar las interacciones de usuario, al tiempo que mantiene una interfaz fluida.

***Javascript* usa un modelo asíncrono y no bloqueante**, con un ***loop* de eventos** implementado en un sólo hilo, (*single thread*) para operaciones de entrada y salida (*input/output*).

Gracias a esta solución, *Javascript* es altamente concurrente a pesar de emplear un sólo hilo.



Antes de explicar como funciona el modelo de *JavaScript* es importante entender algunos conceptos:

- Procesamiento *Single thread* (Hilo único) y *Multi thread* (Hilos múltiples).
- Operaciones de *CPU* y Operaciones de *I/O* (Entrada y Salida).
- Operaciones Concurrentes y Paralelas.
- Operaciones Bloqueantes y No Bloqueantes.
- Operaciones Síncronas y Asíncronas.

## Single thread y Multi thread

Un hilo la unidad básica de ejecución de un proceso, cada que abres un programa como el navegador o tu editor de código, se levanta un proceso en tu computadora e internamente este puede tener uno o varios hilos (*threads*) ejecutándose para que el proceso funcione.

## Operaciones de CPU y de Entrada y Salida

- Operaciones *CPU*: Aquellas que pasan el mayor tiempo consumiendo Procesos del *CPU*, por ejemplo, la escritura de ficheros.
- Operaciones de Entrada y Salida: Aquellas que pasan la mayor parte del tiempo esperando la respuesta de una petición o recurso, como la solicitud a una *API* o *BD*.

## Concurrencia y Paralelismo

- Concurrencia: cuando dos o más tareas progresan simultáneamente.
- Paralelismo: cuando dos o más tareas se ejecutan, al mismo tiempo.

## Bloqueante y No Bloqueante

Se refiere a como la fase de espera de las operaciones afectan a nuestra aplicación:

- Bloqueante: Son operaciones que no devuelven el control a nuestra aplicación hasta que se ha completado. Por tanto el *thread* queda bloqueado en estado de espera.
- No Bloqueante: Son operaciones que devuelven inmediatamente el control a nuestra aplicación, independientemente del resultado de esta. En caso de que se haya completado, devolverá los datos solicitados. En caso contrario (si la operación no ha podido ser satisfecha) podría devolver un código de error.

## Síncrono y Asíncrono

Se refiere a ¿cuándo tendrá lugar la respuesta?:

- Síncrono: La respuesta sucede en el presente, una operación síncrona esperará el resultado.
- Asíncrono: La respuesta sucede a futuro, una operación asíncrona no esperará el resultado.

Con lo anterior en *JavaScript* podemos tener:

- Código síncrono y bloqueante o
- Código asíncrono y no bloquenante

---

## JavaScript Síncrono

Cada operación se hace de una vez, bloqueando el flujo de ejecución, el hilo es bloqueado mientras espera la respuesta, cuando esta se procesa pasa a la siguiente operación y así sucesivamente al terminar todas las operaciones.

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

function dos() {
    console.log("Dos");
}

function uno() {
    console.log("Uno");
    dos();
    console.log("Tres");
}

uno();
console.log("Fin");
```

El resultado en consola es:

Inicio  
Uno  
Dos  
Tres  
Fin

---

# JavaScript Asíncrono

Cada operación se ejecuta y devuelve inmediatamente el control al hilo, evitando el bloqueo, cuando cada operación termine se enviará una notificación de que ha terminado, es entonces cuando la respuesta se encolará para ser procesada.

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

function dos() {
  setTimeout(function () {
    console.log("Dos");
  }, 1000);
}

function uno() {
  setTimeout(function () {
    console.log("Uno");
  }, 0);
  dos();
  console.log("Tres");
}

uno();
console.log("Fin");
```

El resultado en consola es:

Inicio  
Tres  
Fin  
Uno  
Dos

---

[Ver Video](#)

# Mecanismos asíncronos en JavaScript

Para controlar la asincronía, *JavaScript* cuenta con algunos mecanismos:

- **Callbacks.**
- **Promises.**
- **Async / Await.**

## Callbacks

Una **callback** (llamada de vuelta) es una función que se ejecutará después de que otra lo haga.

Es un mecanismo para asegurar que cierto código no se ejecute hasta que otro haya terminado.

Al ser *JavaScript* un lenguaje orientado a eventos, las **callbacks** son una buena técnica para controlar la asincronía, sin embargo... *Callback Hell* 🤪.

```
function cuadradoCallback(value, callback) {
  ... setTimeout(() => {
    ...   callback(value, value * value);
    ... }, 0 | (Math.random() * 1000));
}

cuadradoCallback(0, (value, result) => {
  ... console.log("Inicia Callback");
  ... console.log(`Callback: ${value}, ${result}`);
  cuadradoCallback(1, (value, result) => {
    ...   console.log(`Callback: ${value}, ${result}`);
    cuadradoCallback(2, (value, result) => {
      ...     console.log(`Callback: ${value}, ${result}`);
      cuadradoCallback(3, (value, result) => {
        ...       console.log(`Callback: ${value}, ${result}`);
        cuadradoCallback(4, (value, result) => {
```

```
    console.log(`Callback: ${value}, ${result}`);
    cuadradoCallback(5, (value, result) => {
        console.log(`Callback: ${value}, ${result}`);
        console.log("Fin Callback");
        console.log("Callback Hell !!!!!😺😹");
        console.log("http://callbackhell.com/");
    });
});
```

[Ver Video](#)

# Promises

Una **promesa** es un objeto que representa el resultado de una operación asíncrona y tiene 3 estados posibles:

1. Pendiente.
  2. Resuelta.
  3. Rechazada.

Tienen la particularidad de que se pueden encadenar (*then*), siendo el resultado de una promesa, los datos de entrada de otra posible función.

Las **promesas** mantienen un código más legible y mantenible que las *callbacks*, además tienen un mecanismo para la detección de errores (**catch**) que es posible usar en cualquier parte del flujo de datos.

```
function cuadradoPromise(value) {
  if (typeof value !== "number") {
    return Promise.reject(
      `Error, el valor "${value}" ingresado no es un número`);
  }
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve({
        value,
        cuadrado: value * value
      });
    }, 2000);
  });
}
```

```
        result: value * value,  
        ...});  
    ...}, 0 | (Math.random() * 1000));  
...});  
}  
  
cuadradoPromise(0)  
  .then((obj) => {  
    ... //console.log(obj);  
    ... console.log("Inicio Promise");  
    ... console.log(`Promise: ${obj.value}, ${obj.result}`);  
    ... return cuadradoPromise(1);  
  })  
  .then((obj) => {  
    ... console.log(`Promise: ${obj.value}, ${obj.result}`);  
    ... return cuadradoPromise(2);  
  })  
  .then((obj) => {  
    ... console.log(`Promise: ${obj.value}, ${obj.result}`);  
    ... return cuadradoPromise(3);  
  })  
  .then((obj) => {  
    ... console.log(`Promise: ${obj.value}, ${obj.result}`);  
    ... return cuadradoPromise("4");  
  })  
  .then((obj) => {  
    ... console.log(`Promise: ${obj.value}, ${obj.result}`);  
    ... return cuadradoPromise(5);  
  })  
  .then((obj) => {  
    ... console.log(`Promise: ${obj.value}, ${obj.result}`);  
    ... console.log("Fin Promise");  
  })  
  .catch((err) => console.error(err));
```

[Ver Video](#)

# Async / Await

Las promesas fueron una gran mejora respecto a las *callbacks* para controlar la asincronía en *JavaScript*, sin embargo pueden llegar a ser muy verbosas a medida que se requieran más y más métodos *.then()*.

Las funciones asíncronas (**async / await**) surgen para simplificar el manejo de las promesas.

La palabra **async** declara una función como asíncrona e indica que una promesa será automáticamente devuelta.

Podemos declarar como **async** funciones con nombre, anónimas o funciones flecha.

La palabra **await** debe ser usado siempre dentro de una función declarada como **async** y esperará de forma asíncrona y no bloqueante a que una promesa se resuelva o rechace.

```
function cuadradoPromise(value) {
  if (typeof value !== "number") {
    return Promise.reject(
      `Error, el valor "${value}" ingresado no es un número`
    );
  }

  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve({
        value,
        result: value * value,
      });
    }, 0 | (Math.random() * 1000));
  });
}

async function funcionAsincronaDeclarada() {
  try {
    console.log("Inicio Async Function");

    let obj = await cuadradoPromise(0);
    console.log(`Async Function: ${obj.value}, ${obj.result}`);

    obj = await cuadradoPromise(1);
    console.log(`Async Function: ${obj.value}, ${obj.result}`);

    obj = await cuadradoPromise(2);
    console.log(`Async Function: ${obj.value}, ${obj.result}`);

    obj = await cuadradoPromise("3");
    console.log(`Async Function: ${obj.value}, ${obj.result}`);
  }
}
```

```
...     obj = await cuadradoPromise(4);
...     console.log(`Async Function: ${obj.value}, ${obj.result}`);

...     obj = await cuadradoPromise(5);
...     console.log(`Async Function: ${obj.value}, ${obj.result}`);

...     console.log("Fin Async Function");
} catch (err) {
    console.error(err);
}
}

funcionAsincronaDeclarada();

const funcionAsincronaExpresada = async () => {
try {
    console.log("Inicio Async Function");

    let obj = await cuadradoPromise(6);
    console.log(`Async Function: ${obj.value}, ${obj.result}`);

    obj = await cuadradoPromise(7);
    console.log(`Async Function: ${obj.value}, ${obj.result}`);

    obj = await cuadradoPromise(8);
    console.log(`Async Function: ${obj.value}, ${obj.result}`);

    obj = await cuadradoPromise("9");
    console.log(`Async Function: ${obj.value}, ${obj.result}`);

    obj = await cuadradoPromise(10);
    console.log(`Async Function: ${obj.value}, ${obj.result}`);

    console.log("Fin Async Function");
} catch (err) {
    console.error(err);
}
};

funcionAsincronaExpresada();
```

[Ver Video](#)

Si te sirvió mi contenido, compártelo .



## La lista de MirCha

Una vez al mes envío el contenido que escribo, grabo o produzco, esperando pueda aportar valor personal o profesional.

Puedes darte de baja en cualquier momento, sin hacer preguntas.

¿Te animas?

[SÍ, me suscribo](#)

### TAMBIÉN EN JONMIRCHA

hace 8 meses • 6 comentarios

hace 5 meses • 2 comentarios

hace

**JavaScript****Estoicismo****S  
A**[Comentarios](#)[Comunidad](#) [Políticas de Privacidad](#) [1 Iniciar sesión](#) [Recomendar 6](#) [Tweet](#) [Compartir](#)

**Si te gusta y te sirve mi contenido**

# ¡Apóyame!

Paso la mayor parte de mi tiempo haciendo cursos y tutoriales **gratuitos** sobre desarrollo web.

Únete a las **Membresías** de mi canal de *YouTube* donde tengo diferentes **recompensas** para los que me apoyan.

También puedes invitarme un taco por *Paypal* o hacer una donación a mi cuenta de banco.

**¡Muchas Gracias!**

**Paypal****YouTube**

## Citibanamex