# **Callback Hell**

*Una guía para escribir programas asíncronos en JavaScript*

¿Qué es el " callback hell"?

JavaScript asíncrono, o JavaScript que utiliza callbacks, es difícil de entender intuitivamente. Mucho código acaba teniendo este aspecto:



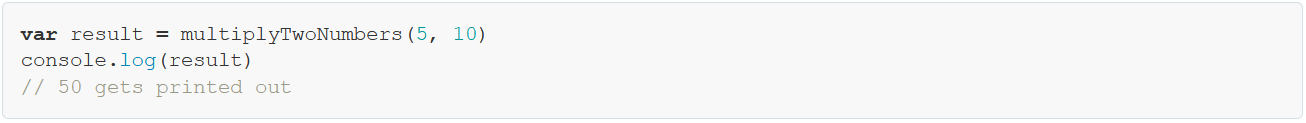
¿Ves la forma de pirámide y todos los }) al final? ¡Vaya! Esto se conoce cariñosamente como callback hell.

La causa del callback hell es cuando la gente trata de escribir JavaScript de manera que la ejecución ocurra visualmente de arriba hacia abajo. Mucha gente comete este error. En otros lenguajes como C, Ruby o Python existe la expectativa de que cualquier cosa que ocurra en la línea 1 terminará antes de que el código de la línea 2 comience a ejecutarse y así sucesivamente en el archivo. Como aprenderás, JavaScript es diferente.

# ¿Qué son las callbacks?

Los callbacks son sólo el nombre de una convención para el uso de funciones de JavaScript. No hay una cosa especial llamada 'callback' en el lenguaje JavaScript, es sólo una convención. En lugar de devolver inmediatamente algún resultado como la mayoría de las funciones, las funciones que utilizan callbacks tardan algún tiempo en producir un resultado. La palabra "asíncrono", significa simplemente que "tarda un poco" o que "ocurre en el futuro, no ahora mismo". Normalmente los callbacks sólo se usan cuando se hace I/O, por ejemplo, descargando cosas, leyendo archivos, hablando con bases de datos, etc.

Cuando llamas a una función normal puedes usar su valor de retorno:

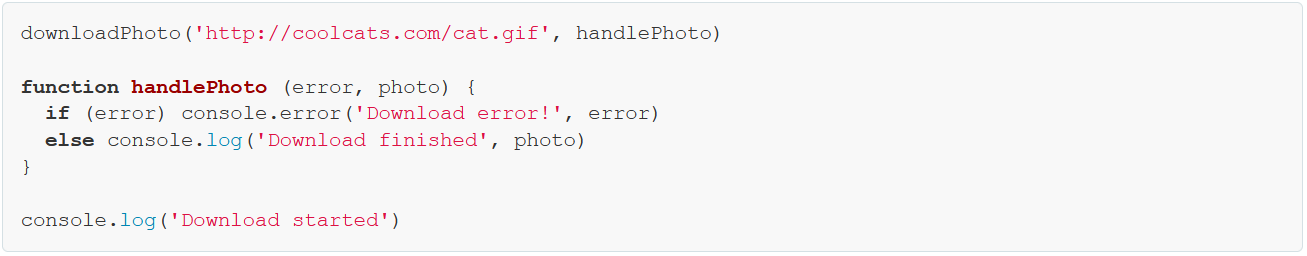


Sin embargo, las funciones que son asíncronas y utilizan callbacks no devuelven nada inmediatamente.



En este caso, el archivo .gif puede tardar mucho tiempo en descargarse, y no quieres que tu programa se detenga (es decir, se "bloquee") mientras esperas a que termine la descarga.

En su lugar, almacena el código que debe ejecutarse después de la descarga en una función. ¡Esta es el callback! Se lo das a la función downloadPhoto y ésta ejecutará tu callback (por ejemplo, te llamará más tarde) cuando se complete la descarga, y pasará la foto (o un error si algo salió mal).



Tenga en cuenta que handlePhoto no se invoca todavía, sólo se crea y se pasa como un callback en downloadPhoto. Pero no se ejecutará hasta que downloadPhoto termine de hacer su tarea, lo que podría llevar mucho tiempo dependiendo de la velocidad de la conexión a Internet.

Este ejemplo pretende ilustrar dos conceptos importantes:

* El callback de handlePhoto es sólo una forma de almacenar algunas cosas para hacerlas en un momento posterior
* El orden en el que ocurren las cosas no se lee de arriba a abajo, sino que salta en función de cuándo se completan las cosas

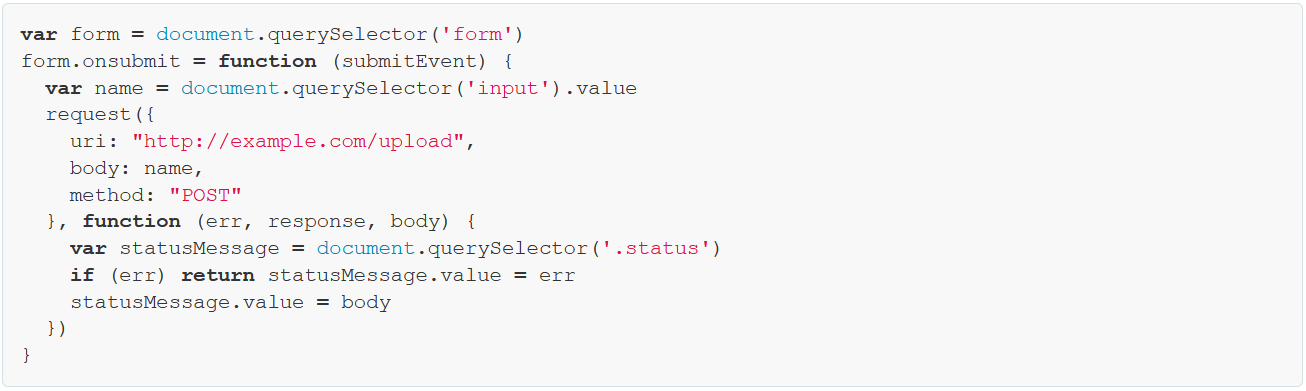
# ¿Cómo puedo arreglar el callback hell?

El callback hell está causado por malas prácticas de codificación. Por suerte, escribir un mejor código no es tan difícil.

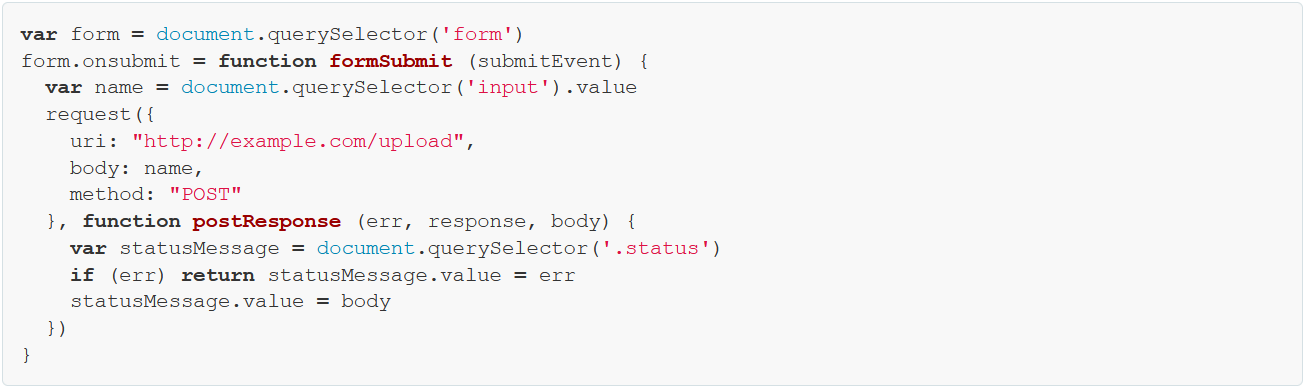
Sólo tienes que seguir tres reglas:

## **1. Mantén tu código superficial**

Aquí hay un poco de JavaScript de navegador desordenado que usa [browser-request](https://github.com/iriscouch/browser-request) para hacer una petición AJAX a un servidor:



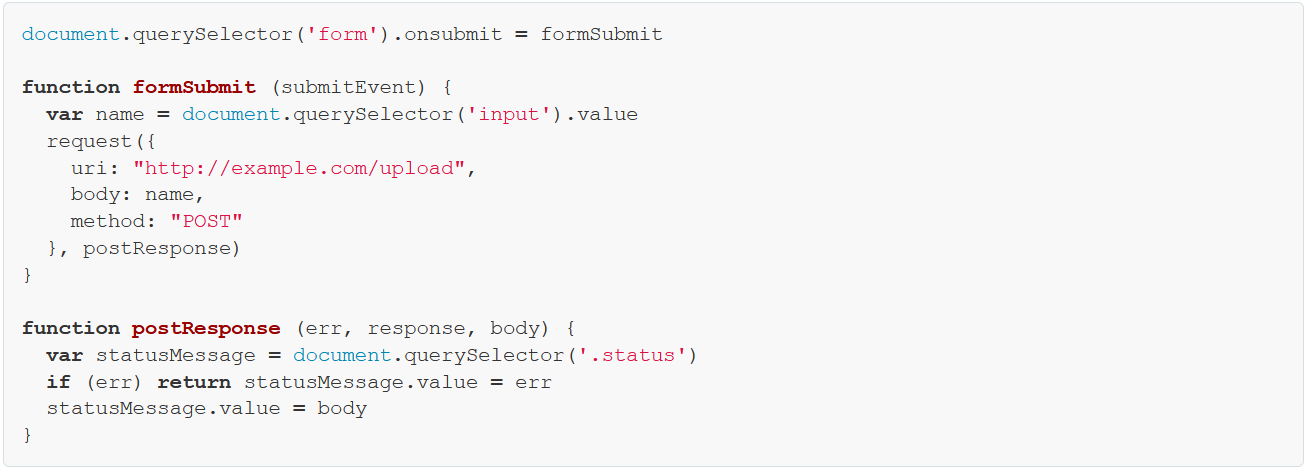
Este código tiene dos funciones anónimas. ¡Démosles nombres!



Como puedes ver nombrar las funciones es súper fácil y tiene algunos beneficios inmediatos:

* hace que el código sea más fácil de leer gracias a los nombres descriptivos de las funciones
* cuando ocurran excepciones, obtendrás stacktraces que hacen referencia a nombres de funciones reales en lugar de "anónimas
* permite mover las funciones y referenciarlas por sus nombres

Ahora podemos mover las funciones a la parte superior de nuestro programa:



Obsérvese que las declaraciones de las funciones están definidas en la parte inferior del archivo. Esto es gracias al [hoisting de funciones](https://gist.github.com/maxogden/4bed247d9852de93c94c).

## **2. Modularizar**

Esta es la parte más importante: Cualquiera es capaz de crear módulos (también conocidos como bibliotecas). Citando a Isaac Schlueter (del proyecto node.js): *"Escribe pequeños módulos que hagan cada uno una cosa, y ensámblalos en otros módulos que hagan una cosa mayor. No puedes entrar en el callback hell si no vas hacia allí"*.

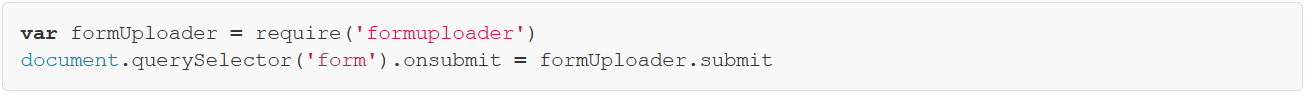
Saquemos el código repetitivo de arriba y convirtámoslo en un módulo dividiéndolo en un par de archivos. Mostraré un patrón de módulo que funciona tanto para el código del navegador como para el código del servidor (o código que funciona en ambos):

Aquí hay un nuevo archivo llamado formuploader.js que contiene nuestras dos funciones de antes:



El bit module.exports es un ejemplo del sistema de módulos de node.js que funciona en node, Electron y el navegador usando [browserify](https://github.com/browserify/browserify). Me gusta bastante este estilo de módulos porque funciona en todas partes, es muy sencillo de entender y no requiere complejos archivos de configuración o scripts.

¡Ahora que tenemos formuploader.js (y que se carga en la página como una etiqueta de script después de ser browserify) sólo tenemos que requerirlo y usarlo! Así es como se ve ahora el código específico de nuestra aplicación:



Ahora nuestra aplicación tiene sólo dos líneas de código y tiene los siguientes beneficios

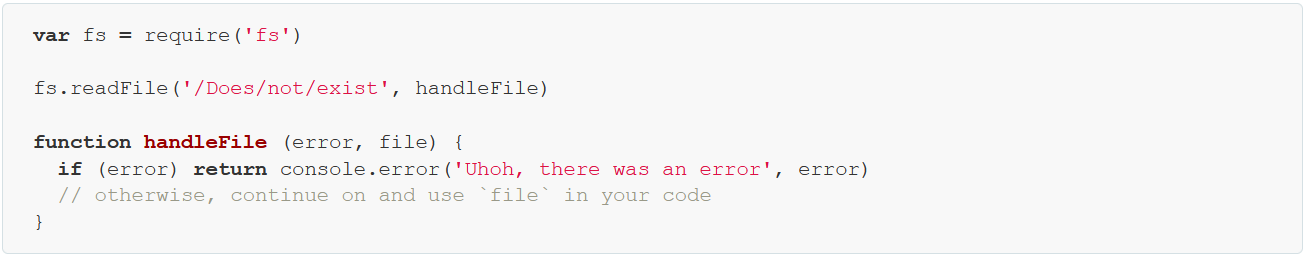
* es más fácil de entender para los nuevos desarrolladores - no se verán atascados por tener que leer todas las funciones de formuploader
* formuploader puede ser utilizado en otros lugares sin duplicar el código y puede ser fácilmente compartido en github o npm

## **3. Manejar cada uno de los errores**

Hay diferentes tipos de errores: errores de sintaxis causados por el programador (normalmente se detectan cuando se intenta ejecutar el programa por primera vez), errores de ejecución causados por el programador (el código se ejecuta pero tiene un error que hace que algo se estropee), errores de plataforma causados por cosas como permisos de archivo no válidos, fallos del disco duro, falta de conexión de red, etc. Esta sección sólo pretende abordar esta última clase de errores.

Las dos primeras reglas tratan principalmente de hacer que tu código sea legible, pero ésta trata de hacer que tu código sea estable. Cuando se trata de callbacks se trata por definición de tareas que se despachan, van y hacen algo en segundo plano, y luego se completan con éxito o se abortan debido a un fallo. Cualquier desarrollador experimentado te dirá que nunca puedes saber cuándo se producen estos errores, así que tienes que planificar que siempre ocurrirán.

Con callbacks, la forma más popular de manejar los errores es el estilo de Node.js, donde el primer argumento de la devolución de llamada siempre se reserva para un error.



Hacer que el primer argumento sea el error es una simple convención que le anima a recordar que debe manejar sus errores. Si fuera el segundo argumento, podría escribir código como function handleFile (file) { } e ignorar más fácilmente el error.

Los linters de código también pueden ser configurados para ayudarle a recordar el manejo de los errores de devolución de llamada. El más simple de usar se llama estándar. Todo lo que tienes que hacer es ejecutar $ standard en tu carpeta de código y te mostrará cada callback en tu código con un error no manejado.

## **Resumen**

1. No anides funciones. Dales nombres y colócalas en el nivel superior de tu programa
2. Utiliza la elevación de funciones a tu favor para mover las funciones "debajo del pliegue".
3. Maneje cada error en cada una de sus devoluciones de llamada. Utiliza un linter como el estándar para ayudarte con esto.
4. Crea funciones reutilizables y colócalas en un módulo para reducir la carga cognitiva necesaria para entender tu código. Dividir tu código en pequeños trozos como este también te ayuda a manejar los errores, escribir pruebas, te obliga a crear una API pública estable y documentada para tu código, y te ayuda con la refactorización.

El aspecto más importante para evitar el infierno de las devoluciones de llamada es mover las funciones fuera del camino para que el flujo de los programas pueda ser más fácilmente entendido sin que los recién llegados tengan que vadear a través de todos los detalles de las funciones para llegar a la carne de lo que el programa está tratando de hacer.

Puedes empezar moviendo las funciones al final del archivo, luego pasarlas a otro archivo que cargues usando un require relativo como require('./photo-helpers.js') y finalmente moverlas a un módulo independiente como require('image-resize').

Estas son algunas reglas generales para crear un módulo:

* Comience por mover el código utilizado repetidamente en una función
* Cuando tu función (o un grupo de funciones relacionadas con el mismo tema) sea lo suficientemente grande, muévelas a otro archivo y expónlas usando module.exports. Puedes cargar esto usando un require relativo
* Si tienes un código que puede ser utilizado en múltiples proyectos dale su propio readme, tests y package.json y publícalo en github y npm. ¡Hay demasiados beneficios impresionantes para este enfoque específico para enumerar aquí!
* Un buen módulo es pequeño y se centra en un problema
* Los archivos individuales de un módulo no deberían ser más largos que unas 150 líneas de JavaScript
* Un módulo no debería tener más de un nivel de carpetas anidadas llenas de archivos JavaScript. Si lo tiene, probablemente esté haciendo demasiadas cosas
* Pida a los programadores más experimentados que conozca que le muestren ejemplos de buenos módulos hasta que tenga una buena idea de cómo son. Si tardas más de unos minutos en entender lo que ocurre, probablemente no sea un buen módulo.

## **Más lecturas**

Intenta leer mi introducción más larga a las devoluciones de llamada, o prueba algunos de los tutoriales de nodeschool.

También revisa el browserify-handbook para ver ejemplos de cómo escribir código modular.

## **¿Qué pasa con las promesas/generadores/ES6, etc.?**

Antes de buscar soluciones más avanzadas, recuerda que los callbacks son una parte fundamental de JavaScript (ya que son sólo funciones) y deberías aprender a leerlas y escribirlas antes de pasar a características más avanzadas del lenguaje, ya que todas ellas dependen de la comprensión de los callbacks. Si aún no puedes escribir código de callbacks mantenible, ¡sigue trabajando en ello!

Si realmente quieres que tu código asíncrono se lea de arriba a abajo, hay algunas cosas elegantes que puedes probar. Ten en cuenta que éstas pueden introducir problemas de rendimiento y/o de compatibilidad en tiempo de ejecución entre plataformas, así que asegúrate de investigar.

**Las promesas** son una forma de escribir código asíncrono que sigue pareciendo que se está ejecutando de forma descendente, y maneja más tipos de errores debido a que se fomenta el uso del manejo de errores estilo try/catch.

**Los generadores** te permiten "pausar" funciones individuales sin pausar el estado de todo el programa, lo que a costa de un código ligeramente más complejo de entender permite que tu código asíncrono parezca ejecutarse de forma descendente. Mira [watt](https://github.com/mappum/watt) para ver un ejemplo de este enfoque.

**Las funciones asíncronas** son una característica propuesta de ES7 que envolverá aún más los generadores y las promesas en una sintaxis de mayor nivel. Compruébalo si te parece interesante.

Personalmente uso callbacks para el 90% del código asíncrono que escribo y cuando las cosas se complican traigo algo como [run-parallel](https://github.com/feross/run-parallel) o [run-series](https://github.com/feross/run-series). No creo que los callbacks vs promesas vs cualquier otra cosa realmente hagan una diferencia para mí, el mayor impacto viene de mantener el código simple, no anidado y dividido en pequeños módulos.

Independientemente del método que elijas, siempre maneja cada error y mantén tu código simple.

**Recuerda, sólo tú puedes evitar el callback hell de llamada y los incendios forestales.**

Puedes encontrar la fuente de esto en [github](https://github.com/maxogden/callback-hell).