Métodos de procesamiento asíncrono

Fuente: <https://www.youtube.com/watch?v=Q3HtXuDEy5s>

En este apunte vamos a ver los tres metodos de procesamiento asíncrono que soporta NodeJS

* callbacks
* promesas
* AsyncAwait

Los **callbacks** son la forma genérica de nodejs para llamar a procesos y seguir su ejecución sin tener que esperar que estos terminen. Tienen como desventaja que al anidar demasiados el código se torna ilegible (callback hell). Para solucionarlo con **EcmaScript 6** se introdujeron las **promesas**, que representaron una forma mas prolija de escribir funciones asíncronas. Por último **Async Await** es el método mas moderno de operar y tiene como principal ventaja que el código si bien sigue siendo asíncrino, la sintaxis se asemeja mucho mas a código síncrono y permite ordenar de mejor manera el código ya se se almacenan en variables el resultado de las llamadas a funciones.

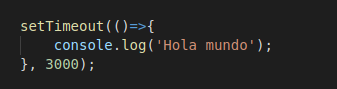
Vamos a ver con un ejemplo no funcional de conexión a base de datos cómo sería la diferencia entre cada tipo de proceso, y luego un ejemplo utilizando la api de github mediante el módulo node-fetch.

## 1 – Callbacks

Como ya vimos en otros apuntes, NodeJs trabaja generalmente de manera asíncrona mediante callbacks. Estas son funciones que se ejecutan luego de terminar cierto proceso sin necesidad de bloquear la aplicación esperando a que este se ejecute. Los callbacks son funciones que se pasan como parámetros a otras funciones.

Por ejemplo en la funcion **setTimeout**, esta recibe como parámetros

* qué hacer (en forma de callback)
* cuánto tiempo esperar.



callback

Esta funcion muestra un mensaje por consola pero luego de que pasen 3 segundos

Ahora veamos un ejemplo mas complejo. En una aplicación lo mas común es hacer consultas a una base de datos, esperar su respuesta y luego hacer algo con esta información, como por ejemplo enviarla al navegador. Con NodeJS las consultas se realizan de manera asíncrona, de maner que el servidor puede seguir recibiendo peticiones mientras espera la respuesta de la base de datos.

Supongamos que en el modelo de datos tenemos una colección de personas y otra de tareas, donde cada persona es responsable de una tarea. Si queremos indicar que una persona completó su tarea, el código usando callbacks sería algo así.



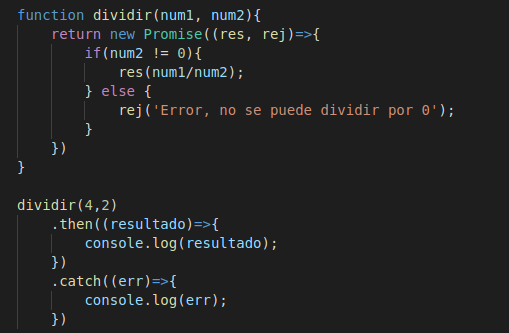
Esta consulta si bien es sencilla ya presenta varios inconvenientes. Principalmente la dificultad de lectura, y además si se lo analiza podemos notar que todos los errores se están manejando de la misma manera.

Para solucionar estos problemas puede realizarse la misma consulta utilizando promesas.

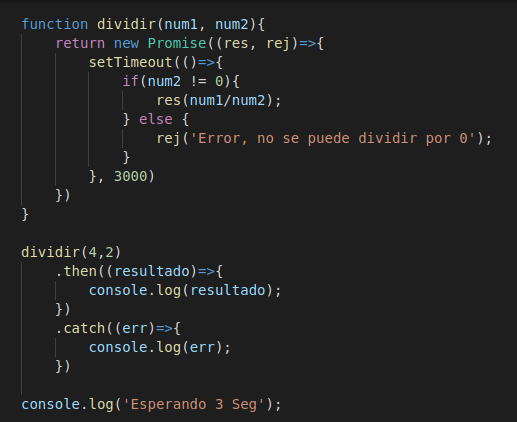
## 2 - Promesas

Las promesas son una de las características introducidas en EcmaScript6, y son una forma mas prolija de manejar código asíncrono. Estas funcionan de la siguiente manera, dentro de una función puede declararse una promesa, y esta recibe dos parámetros, generalmente llamados **res** (response) y **rej** (reject). Ambos son devueltos en la función, y al invocarla deben ser gestionados con los métodos **then** y **catch**.

Es mas fácil verlo con un ejemplo. Si queremos un función que divida dos números y controle que no se pueda dividrir por 0 podemos hacerlo de la siguiente manera.



En este caso la operación es demasiado sencilla. Pero la idea es usar las promesas en procesamiento asíncro, entonces le ponemos un timeOut a la función dividir para simular un tiempo de procesamiento considerable.



Ahora vemos que al ejecutarlo sale primero el mensaje y después el resultado de la operación.

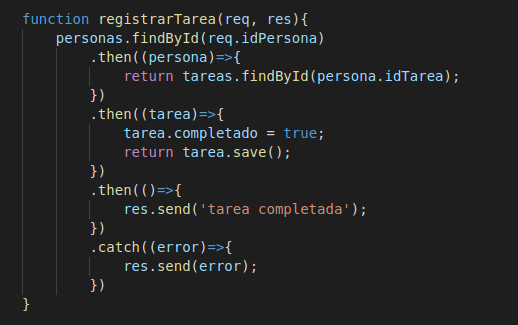


Volviendo al ejemplo de la base de datos, vimos que la funcion findById requería como parámertos el id, y un callback que recibia los parámetros err y el objeto donde guardar las respuesta



Esto es porque internamente trabaja con promesas, entonces hay una correlacion entre el error con rej y la persona con res.

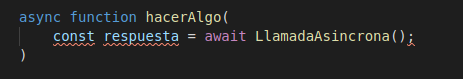
Entonces la consulta anterior puede reescribirse usando promesas de la siguiente manera.



Veamos que cada vez que se llama a una función que trabaja de manera asíncrona (findById, save), esta o bien es manejada directamente con el método **then()**, o si se encuentra dentro de otra llamada a esta misma función retorna la respuesta a un nivel superior para ser gestionada por el siguiente then. También podemos ver que las respuestas al cliente, tanto si se completó la operación como si hubo algún error, se envían solo una vez, con lo que no se repite código de manera innecesaria. Además si hay algún error se maneja de la misma forma que las excepciones, pasandolo a niveles superiores hasta encontrar un método **catch()** que lo resuleva.

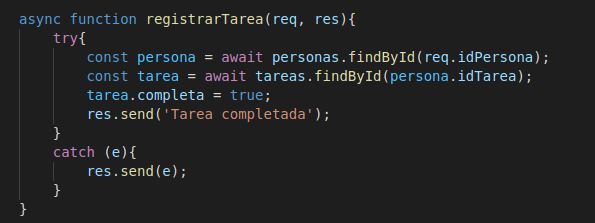
## 3 – Async Await

Async Await es una nueva forma de manejar llamadas a funciones asíncronas que presenta una gran ventaja sobre las promesas ya que está implementado de tal manera que se asemeja al código síncrono tradicional. La sintaxis básica es.



Otra ventaja es que los errores se manejan con las tradicionales clausulas try/catch, por lo que se facilita la transferencia si uno está acostumbrado a trabajar con lenguajes asíncronos como por ejemplo Java.

Volviendo al ejemplo de la consulta el código ahora sería el siguiente.



## 4 – Ejemplo API

Ahora veamos un ejemplo de como se utiliza la API de github con cada uno de los tres métodos.

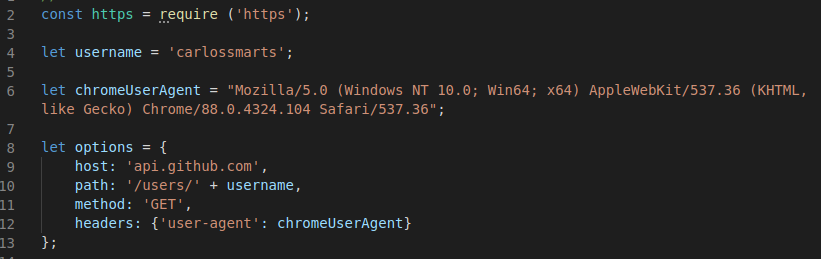
**Callbacks**

En este caso, para consumir el servicio de Github usamos el módulo de npm **https**. Para usarlo lo instalamos con

npm install htpps – save

y el código para realizar la petición es el siguiente:

1) Importamos el módulo https e indicamos mediante variables el nombre de usuario de git, un agente de usuario (necesario para simular que la petición se hace desde un navegador, en este caso Chrome), y las opciones necesarias para realizar la petición.



2) Para realizar la petición usamos el metodo **request()** del módulo https, este recibe como parámetros las opciones declaradas previamente, y un callback con un objeto response donde se va a guardar la respuesta. Dentro del callback se escuchan los eventos que indican que se retorno una respuesta (**‘data’**) y que finalizó la petición (**‘end’**). Finalmente, antes de cerrar el objeto request se verifica si hubo algún error con el evento **‘error’.**



Al ejecutar el archivo se muestra por consola el nombre del usuario de github.



**Promesas**

Al utilizar promesas podemos hacer uso de un módulo de npm que permite hacer peticiones http de una forma mucho mas práctica llamado **node-fetch**. Como todo módulo de terceros, lo primero que hay que hacer es instalarlo

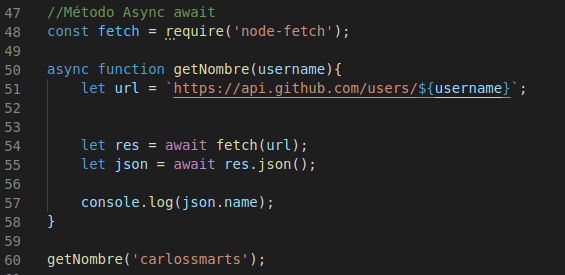
npm install node-fetch - - save

Ahora el código es mucho mas corto, primero porque no hay que setear tantas opciones, y segundo porque las respuestas se manejan con promesas de manera mas sencilla. (de hecho ni siquiera requere una explicación).



**Async Await**

Con Async Await el proceso es bastante similar, solo que requiere que implementar una función.



Pero con este método se puede ir aún mas alla. Ya que una función asíncrona también trabaja con promesas, en lugar de imprimir por consola dentro de la misma podemos retornar una respuesta o arrojar excepciones y manejarlas con un bloque try catch. Esto puede hacerse con promesas

|  |  |
| --- | --- |
|  |  |
|  |  |

O con una **función autoejecutable**

