

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.

```
setTimeout(()=>{  
  console.log('Hola mundo');  
}, 3000);
```



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 manera 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.

```
function registrarTarea(req, res){
  personas.findById(req.userId, (err, persona)=>{
    if (err){
      res.send(err);
    } else{
      tareas.findById(persona.idTarea, (err,tarea)=>{
        if (err){
          res.send(err);
        } else{
          tarea.completado = true;
          tarea.save((err)=>{
            if(err){
              res.send(err);
            } else{
              res.send('Tarea Completada');
            }
          })
        }
      })
    }
  })
}
```

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 una función que divida dos números y controle que no se pueda dividir por 0 podemos hacerlo de la siguiente manera.

```
function dividir(num1, num2){
  return new Promise((res, rej)=>{
    if(num2 !== 0){
      res(num1/num2);
    } else {
      rej('Error, no se puede dividir por 0');
    }
  })
}

dividir(4,2)
  .then((resultado)=>{
    console.log(resultado);
  })
  .catch((err)=>{
    console.log(err);
  })
}
```

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.

```
function dividir(num1, num2){
  return new Promise((res, rej)=>{
    setTimeout(()=>{
      if(num2 !== 0){
        res(num1/num2);
      } else {
        rej('Error, no se puede dividir por 0');
      }
    }, 3000)
  })
}

dividir(4,2)
  .then((resultado)=>{
    console.log(resultado);
  })
  .catch((err)=>{
    console.log(err);
  })

console.log('Esperando 3 Seg');
```

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

```
carlos@carlos-ThinkPad-T430:~/Escritorio/Programación/NodeJS/Proyectos/6 - procesamiento asíncrono$ node promesas.js
Esperando 3 Seg
2
```

Volviendo al ejemplo de la base de datos, vimos que la función `findById` requería como parámetros el id, y un callback que recibía los parámetros `err` y el objeto donde guardar la respuesta

```
personas.findById(req.userId, (err, persona)=>{
```

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

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

```
function registrarTarea(req, res){
  personas.findById(req.idPersona)
    .then((persona)=>{
      return tareas.findById(persona.idTarea);
    })
    .then((tarea)=>{
      tarea.completado = true;
      return tarea.save();
    })
    .then(()=>{
      res.send('tarea completada');
    })
    .catch((error)=>{
      res.send(error);
    })
}
```

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, pasándolo a niveles superiores hasta encontrar un método **`catch()`** que lo resuelve.

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.

```
async function hacerAlgo(
  const respuesta = await LlamadaAsincrona();
)
```

Otra ventaja es que los errores se manejan con las tradicionales cláusulas 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.

```
async function registrarTarea(req, res){
  try{
    const persona = await personas.findById(req.idPersona);
    const tarea = await tareas.findById(persona.idTarea);
    tarea.completa = true;
    res.send('Tarea completada');
  }
  catch (e){
    res.send(e);
  }
}
```

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 https --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  const https = require('https');
3
4  let username = 'carlosmarts';
5
6  let chromeUserAgent = "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML,
  like Gecko) Chrome/88.0.4324.104 Safari/537.36";
7
8  let options = {
9    host: 'api.github.com',
10   path: '/users/' + username,
11   method: 'GET',
12   headers: {'user-agent': chromeUserAgent}
13 };
```

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**'.

```
15 const req = https.request(options, (res)=>{
16   let body = '';
17   res.on('data', (out)=>{
18     body += out;
19   });
20
21   res.on('end', (out)=>{
22     let json = JSON.parse(body);    //información de usuario de github
23     console.log(json.name);        //nombre del usuario
24   });
25 });
26
27 req.on('error', (err)=>{
28   console.log(err);
29 })
30
31 req.end();
```

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

```
carlos@carlos-ThinkPad-T430:~/Escritorio/Programación/NodeJS/Proyecto/6 - procesamiento asíncrono$ node git.js
Carlos Sebastián Martínez
```

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 requiere una explicación).

```
//Método Promesas
const fetch = require('node-fetch');
let username = 'carlossmarts';
let url = `https://api.github.com/users/${username}`;

fetch(url)
  .then((res)=>{
    return res.json();
  })
  .then((json)=>{
    console.log(json.name);
  });
```

Async Await

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

```
47 //Método Async await
48 const fetch = require('node-fetch');
49
50 async function getNombre(username){
51     let url = `https://api.github.com/users/${username}`;
52
53
54     let res = await fetch(url);
55     let json = await res.json();
56
57     console.log(json.name);
58 }
59
60 getNombre('carlossmarts');
```

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

```
getNombre('carlossmarts')
    .then((nombre)=>{
        console.log(nombre);
    })
    .catch((err)=>{
        console.log(`Error: ${err}`);
    });
```

```
carlos@carlos-ThinkPad-T430:~$
Carlos Sebastián Martínez
```

```
getNombre('algunusuarioinexistente')
    .then((nombre)=>{
        console.log(nombre);
    })
    .catch((err)=>{
        console.log(`Error: ${err}`);
    });
```

```
carlos@carlos-ThinkPad-T430:~/Es
Error: Error: no existe usuario
```

O con una **función autoejecutable**

```
(async function (){
    try{
        let nombre = await getNombre('carlossmarts');
        console.log(`Nombre del usuario de git: ${nombre}`);
    } catch (err){
        console.log(`Error: ${err}`);
    }
})();
```

