

NodeJS & GraphQL



Sobre mí

- Alan Berdinelli (<https://alanberdinelli.link>)
- Ex Incluit
- Fullstack Developer (React, NodeJS, TS y AWS)
- Drummer, gamer, nerd

Introducción

Qué es NodeJS? Es un lenguaje? Es una librería? Es un framework? Es un avión? No. Es NodeJS. Es un entorno de ejecución. Basado en el motor de JavaScript de Chrome. Permite ejecutar JavaScript del lado del servidor, es decir, en una máquina en la nube o un servidor dedicado. De forma que los usuarios no puedan acceder al código y aprovechando las ventajas de JS.

Por qué necesitamos Javascript en el servidor? Porque PHP y Java apestan, claro. Nah, hablando seriamente. Javascript no tiene ciertas "bondades" como fuerte tipado. Pero es un lenguaje dedicado a eventos, y es single-threaded.

La flexibilidad que surgió a partir de NodeJS es masiva. Se puede desarrollar aplicaciones para casi cualquier dispositivo. Desktop, mobile, web, TVs, autos, etc.

Proyectos

NodeJS tiene su gestor de paquetes, npm. Tal como para PHP existe composer, para Java Maven o Gradle, o gem para Ruby, con él podemos definir las dependencias de nuestro proyecto, junto con otros detalles de ejecución y también información de su autor. Toda la información del paquete se guarda en un archivo package.json y lógicamente npm expone una CLI para interactuar.

Install

- *Windows*: Instalador, o gestor de paquetes "Chocolatey"
- *Linux*: La mayoría de las distribuciones lo traen disponible en sus repositorios, sino ver en la web de NodeJS cómo agregar los repositorios propios de Node.
- *Mac*: Disponible en Homebrew

JSON (pre-requisito para seguir)

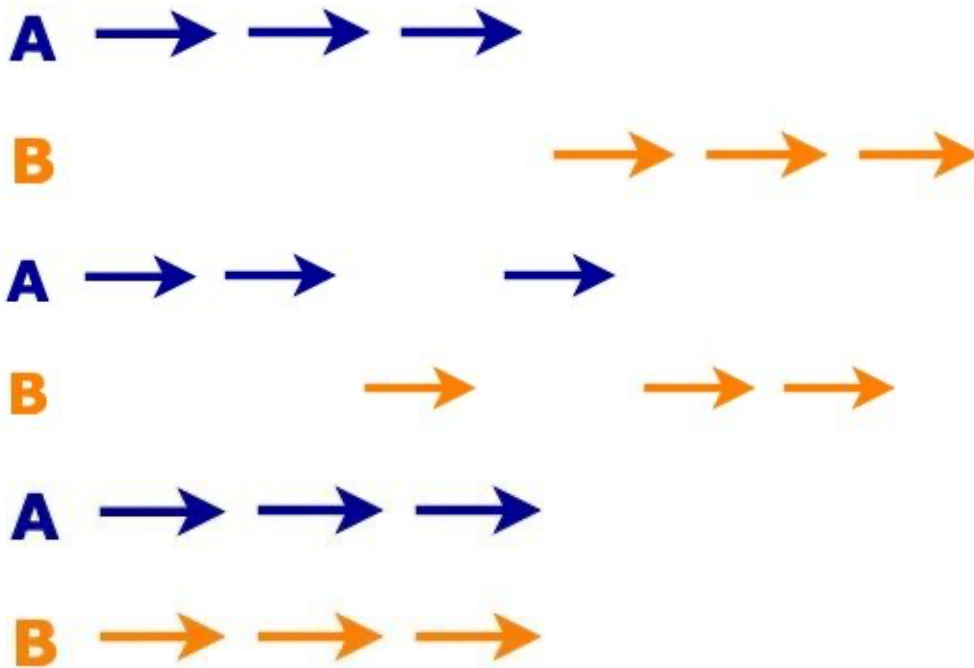
JSON son siglas de JavaScript Object Notation. Ojo! **No es** un objeto JavaScript! Es una forma de escribir información, específicamente dentro de texto. En otras palabras, JSON puede considerarse un string. Lo que normalmente hacemos es interpretar ese string para transformarlo en un objeto en memoria utilizable por el lenguaje.

Concurrencia y paralelismo

Empecemos por lo básico, que son cada una de ellas y cuál es la diferencia? Existen diferentes tipos de flujos a la hora de procesar datos o algoritmos. Entre ellas las más comunes son de forma, concurrente, paralelas o secuenciales. Suponiendo que tenemos dos métodos A y B ejecutándose que realizan cosas independientes.Cuál es la forma en la que podríamos ejecutarlos?

- **Paralelo:** Ejecutamos ambos al mismo tiempo.
- **Secuencial:** Primero ejecutamos A, luego B.
- **Concurrente:** Ejecutamos A, pero aunque no haya terminado por completo, en cierto punto de esa ejecución, también empezamos a ejecutar B.

Ejemplo visual: cuál es cual?



Cómo maneja Node estos diferentes tipos de flujo?

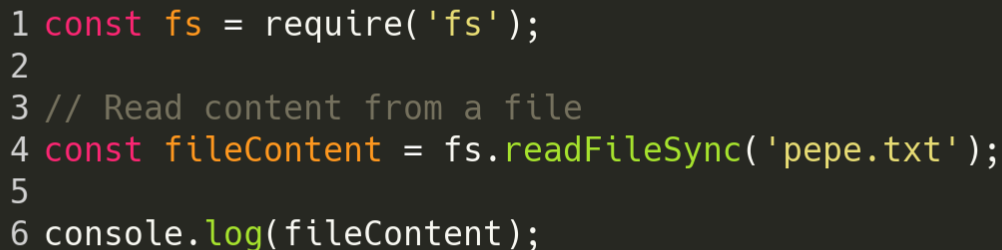
Node tiene una forma muy particular de manejar las operaciones I/O. (Qué significa este término? Simplemente, operaciones de entrada y salida de la aplicación en sí. Por ejemplo, manejo de bases de datos, peticiones HTTP, escritura de archivos, etc). Básicamente, node tiene dos formas de manejarlos, bloqueante y no bloqueante. Tal como sus nombres lo dicen, el bloqueante no va a permitir la ejecución de la siguiente operación mientras que el no bloqueante la permitiría.

Callbacks

Usted está hablando de callbacks? Bueno un poco de esto, un poco de aquello, usted me entiende.

Para empezar, un callback, es una porción de código (una función) que es ejecutada al finalizar una operación no bloqueante. ¿Por qué lo necesitamos? Si una operación es no bloqueante, esto significa que Node, no va a esperar a que la misma termine antes de continuar ejecutando nuestro código, por lo que necesitaríamos alguna forma de saber cuándo este proceso terminó para controlar el resultado, si es que necesitamos utilizarlo.

Proceso bloqueante:



```
1 const fs = require('fs');  
2  
3 // Read content from a file  
4 const fileContent = fs.readFileSync('pepe.txt');  
5  
6 console.log(fileContent);
```

En este caso, leemos el contenido de un archivo pero utilizamos el método `readFileSync` (que es una versión corta de `synchronously` o en español, `sincrónicamente`). El cual lo hace de manera bloqueante. Por lo que nuestro `console.log` no se ejecutaría hasta que el proceso se termine.

Proceso no bloqueante:

```
1 const fs = require('fs');
2
3 // Read content from a file
4 fs.readFile('pepe.txt', function (err, content) {
5   if (err) {
6     throw new Error(err);
7   }
8
9   console.log(content.toString());
10 });
11
12 console.log('I might still be reading');
```

En este otro caso, la operación es asíncrona, es decir no es bloqueante, por lo que ahora necesitamos utilizar un callback, si es que nosotros queremos mostrar el contenido del archivo.

Cómo sabe NodeJS cuando debe ejecutar el callback?

Acá entra en juego uno de los conceptos que más nos ha costado entender a los programadores de Node. Es el acorde con sevilla para quienes han aprendido guitarra. Básicamente, es algo que define qué tanto te gusta Node, dependiendo si lo comprendes o no.

De forma muy simplificada, Node necesita una especie de “secretario”, que se encargue de ir organizando las operaciones. Ya que al existir operaciones asíncronas, se le debe hacer un seguimiento a la misma. Esto es a lo que le llamamos Event-Loop.

Básicamente, el event-loop maneja colas. En particular, hay una cola específica de callbacks, a la que se le agrega el callback de nuestro readFile si tomamos el ejemplo anterior.

Probablemente lo estabas sospechando: esto significa que el callback no se ejecuta inmediatamente al terminarse la operación.

Event-Loop, Callstack, etc

- Tal como lo define su nombre, el event loop es un bucle
- El Event-Loop, tiene “pasos” que va ejecutando en un orden particular, los mismos se denominan “Tick”
- El call stack, es decir, la ejecución de las funciones en nuestro código, también es controlada por este event loop
- Dicho callstack, se agrega como otro paso del event loop
- Cuando el event loop no tiene callbacks por ejecutar, procede a continuar el Stack (de hecho, hace algunos otros pasos, pero eso ya es para nerds)
- El comienzo de ejecución de nuestro código tanto como otras partes de él, son considerados callbacks (o eventos) dentro del event loop, y son agregados a una cola de ejecución
- Corre infinitamente hasta que se termine el proceso

Promesas

Debido a que generalmente al trabajar con node vamos a necesitar hacer varias cosas, donde algunas dependen de otras, y no queremos que nuestro código sea bloqueante, requeriríamos involucrar muchos callbacks. Es decir, tendríamos callbacks dentro de callbacks dentro de callbacks... dentro de callbacks. Además, que pasa si quiero ejecutar dos cosas asíncronas al mismo tiempo? No hay que temer, para eso existen las promesas.

Las promesas son una de las features más utilizadas y dentro de todo recientes que se le agregó a Javascript de forma nativa, previamente para poder “emular” promesas se utilizaban librerías de tercero como por ejemplo, bluebird.

```
1 const fs = require('fs');
2
3 // Read content from a file
4 const readPromise = new Promise((resolve, reject) => {
5   fs.readFile('pepe.txt', function (err, content) {
6     if (err) {
7       return reject(err);
8     }
9
10    return resolve(content.toString());
11  });
12 });
13
14 readPromise
15   .then(content => console.log(content))
16   .catch(err => console.log('Something went wrong'));
```

Tenemos dos principales formas de generar promesas:

La primera y quizás más antigua, es crear una promesa y guardarla en una variable, para luego utilizar sus funciones `.then` y `.catch` que nos permiten setearles un callback para cuando terminen.

```

1 const fs = require('fs');
2
3 // Read content from a file
4 async function readPromise() {
5     return new Promise((resolve, reject) => {
6         fs.readFile('pepe.txt', function (err, content) {
7             if (err) {
8                 return reject(err);
9             }
10
11             return resolve(content.toString());
12         });
13     });
14 }
15
16 try {
17     const content = await readPromise();
18     console.log(content);
19 }
20 catch (err) {
21     console.log('Someting went wrong');
22 }

```

Esta otra forma, nos permite utilizar el operador await, el cual hace que la promesa devuelva inmediatamente el valor que le pasaría como parámetro al callback del .then(). Un tip a tener en cuenta, todas las funciones asíncronas son promesas.

```

1 const fs = require('fs');
2
3 // Sequence
4 promiseThatReturnsAnotherPromise
5     .then(result1 => console.log(result1))
6     .then(result2 => console.log(result2));

```


Al tener la posibilidad de configurar nuestros callbacks de otra manera, podemos pasar a ejecutar código de forma secuencia o paralela con mucha facilidad:

```
1 const fs = require('fs');
2
3 // Parallel
4 const [result1, result2, result3] = await Promise.all([
5   promise1,
6   promise2,
7   promise3
8 ]);
9
10 console.log(result1, result2, result3);
11
```

```
1 const fs = require('fs');
2
3 // Parallel
4 Promise.all([promise1, promise2, promise3])
5   .then(results => {
6     const [result1, result2, result3] = results;
7
8     console.log(result1, result2, result3);
9   });
```

Como pueden ver, los callbacks dentro de callbacks, dentro de callbacks... dentro de callbacks ya no son necesarios. Tenemos la opción tanto de encadenar promesas como de ejecutarlas en paralelo y esperar a que todas terminen antes de proceder a realizar otra acción.

Un tip a tener en cuenta, el operador `await` solamente puede ser utilizado dentro de funciones declaradas con `async`.

Y si, si prestaste atención vas a notar que `Promise.all()` devuelve una promesa 😊.

Actividad

No se preocupen, son cosas muy sencillas, no debería tomarles más de un sprint...

1. Instalar NodeJS/npm
2. Crear un paquete
3. Crear una función que reciba un callback y lo ejecute al finalizar la ejecución. Puede ser cualquier cosa, hasta una cuenta matemática
4. Crear otra función diferente, que transforme la función anterior en una promesa
5. Crear una última función llamando a las dos anteriores y combinando su respuesta