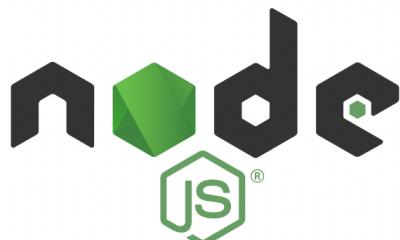


Cómo pasé un proceso en Node.js de 5 horas a 5 minutos

Cómo pasé un proceso en



de 5 horas a 5 minutos



@ulisesantana



Hola! Hoy vengo aquí para hablar de cómo pasé un proceso en Node.js de 5 horas a 5 minutos. Sin embargo, antes me gustaría presentarme.

Ulises Santana

Full Stack Developer en



ulisesantana.dev



@ulisesantana



Soy Ulises Santana y trabajo como Full Stack Developer en Lean Mind donde ayudamos a otras empresas a hacer que sus proyectos de software sean más sostenibles. Soy de Gran Canaria, pero actualmente vivo en El Hierro desde donde trabajo en remoto.



@ulisesantana



Tengo una perrita llamada Mocha (en honor al nombre en clave de JavaScript cuando Brendan Eich empezó a trabajar en él) y un gatete llamado Null.

Antes de explicar cómo pasé un proceso en Node.js de 5 horas a 5 minutos, vamos a empezar por el contexto de esta historia. Estaba trabajando para un cliente en un equipo de 5 personas, en el que las otras 4 se habían incorporado en los últimos 3 meses, mientras que yo llevaba casi un año con el cliente. En este equipo mi rol era el de Senior Node.js Developer y era el único que tenía experiencia previa trabajando con Node.js. Aparte había otra persona con experiencia con JavaScript y Dart, lo cual hacía que le resultara fácil adaptarse a los proyectos en TypeScript, que es el lenguaje en el que estaban todos los proyectos. Sin embargo, las otras tres personas del equipo tenían muy poca experiencia previa en JavaScript.

Por otro lado, estábamos trabajando en las distintas partes de un motor de facturación que necesitaba ser adaptado para un cambio legislativo. Esto último significa que el deadline no se podía mover, si el cambio no estaba hecho para esa fecha la empresa no podía generar la facturación del siguiente mes. En caso de que no llegáramos le rompíamos el cash flow. Suave, sin presión.

Contexto

1. No todo el equipo controlaba la tecnología en la que se estaba trabajando.
2. El deadline es fijo y crítico, ya que rompemos el cash flow de la empresa en caso de retrasarnos.



@ulisesantana



Por concluir esta contextualización:

- No todo el equipo controlaba la tecnología en la que se estaba trabajando.
- El deadline es fijo y crítico, ya que rompemos el cash flow de la empresa en caso de retrasarnos.

Dame 6 horas para cortar un árbol y pasaré 4 afilando el hacha



@ulisesantana



A Abraham Lincoln se le atribuye la siguiente frase: *Dame 6 horas para cortar un árbol y pasaré 4 afilando el hacha*. Sabíamos que iba a haber un cambio legislativo que conllevarían cambios en los

proyectos, así que desde 2 meses antes del deadline propusimos refactorizar partes de los proyectos y uno en concreto solicitamos rehacerlo desde cero, ya que en ese entonces era realmente un prototipo que funcionaba, pero costaba mantener y con el cambio legislativo se iba a hacer más insostenible. Nos dieron luz verde a esta propuesta y dicho prototipo iba a ser rehecho desde cero. Vamos a llamar a este proyecto el Proyecto Leñador.

El Proyecto Leñador



@ulisesantana



En Lean Mind por regla general trabajamos haciendo pair o mob programming, por lo que nadie nunca está solo y así facilitamos que el código sea más sostenible, además de que tanto la autoría del código como el conocimiento se comparta. Sin embargo, como teníamos 5 proyectos que actualizar decidimos dividirnos lo máximo posible para poder abarcar al menos 3 proyectos a la vez y poder tener los cambios lo antes posible. Eso sale a 2 personas por proyecto y una persona sola. Esa persona que se quedó sola fui yo y estuve a varias bandas asistiendo a los diferentes equipos a la par que trabajaba en el proyecto en el que me tocaba.

Esta situación hizo que por falta de tiempo descuidara el proceso de code review y simplemente me centrara en resolver las dudas del equipo sobre todo en dominio, ya que recuerdo que el resto del equipo llevaba sólo 3 meses con el cliente y el dominio del negocio tenía una curva de asimilación de al menos 6 meses. Además, como hacíamos TDD, si los test pasan y reflejaban bien las especificaciones de negocio no había de qué preocuparse.

El Proyecto Leñador fue llevado a cabo por miembros del equipo que no tenían mucha experiencia en JavaScript y ninguna en TypeScript. Esto no suponía a priori ningún problema porque ya llevaban tres meses haciendo pair o mob programming con otros miembros del equipo que sí tenían experiencia y estas mismas personas habían hecho aportaciones a los diferentes proyectos en TypeScript. Simplemente pedían ayuda o consejo cuando lo necesitaban y se les asistía.

La realidad es que el salto de calidad en el Proyecto Leñador era más que evidente. No vi el proyecto en su estado final directamente, sino que lo vi evolucionar a lo largo de las semanas y realmente era mucho más claro en su propósito y no había sorpresas en la implementación. Yo había sido parte del equipo que había hecho ese prototipo 9 meses atrás y la verdad es que había ciertas partes que para mí eran un pelín oscuras, que no terminaba de entender cómo funcionaban o cuál era su propósito final. Esto se debía, entre otras cosas, a que el prototipo original no se había hecho enteramente en mob o pair programming, sino que había partes enteras que habían sido hechas por un desarrollador que ya no formaba parte del equipo.

Volviendo al Proyecto Leñador, estaba muy orgulloso de lo que el equipo había conseguido, realmente era un proyecto mucho más sostenible, eliminando sorpresas. Sin embargo, cuando estaba terminado e hice una última revisión algo más extensa con el equipo veía algunos flujos de datos que tenían toda la pinta de bloquear el Event Loop, o al menos parte de él, pudiendo provocar pérdidas de performance. Para comprobar el performance de este nuevo proyecto pasé al siguiente paso que teníamos planeado: hacer una prueba comparando el prototipo original con el Proyecto Leñador.

Prototipo original:

~7 minutos

Proyecto Leñador:

5 horas 7 minutos y 54 segundos



@ulisesantana



El prototipo original basándose en un set de datos de unos cientos de miles de registros era capaz de hacerlo todo en unos 7 minutos. Con el mismo set de datos probé con el Proyecto Leñador y el resultado fue que tardó nada más y nada menos que **5 horas 7 minutos y 54 segundos**.

+ 4400%



@ulisesantana



Estamos hablando de que tardaba 44 veces más. El proceso real en producción tardaba cada noche unos 40 minutos, por lo que si mandábamos esto a producción el nuevo proceso tardaría unas 29 horas y 20 minutos, cada día. Esta pérdida de performance era inasumible. En ese momento mi yo interno era algo así:



@ulisesantana



Deadline: 10 días

NATURALES



@ulisesantana



Por meter más leña al fuego [¿lo pillas? Leña, Proyecto Leñador 🚧], esto pasó a unos 10 días del deadline, 10 días naturales. No podíamos replantear el proyecto, había que optimizarlo en menos de una semana, además de que había más cosas en la parrilla. Recuerdo que habían otros 4 proyectos que necesitaban ser actualizados para el cambio legislativo. En esta situación, por un lado me motivaba a mí mismo pensando cosas del tipo *Llevo toda mi vida preparándome para este momento, los talleres sobre asincronía en Node.js con Matteo Collina y James Snell van a dar sus frutos*



@matteocollina



@jasnell



@ulisesantana



Por cierto, un saludo desde aquí a Matteo Collina y James Snell. Son dos personas de la comunidad de Node.js de las que he aprendido muchísimo. Aunque ellos no me conozcan los sigo y admiro desde hace un par de años.

Volviendo a la historia, en ese momento otra parte de mí era una mezcla de esto:



@ulisesantana



La realidad es que entré en modo pánico y empecé a refactorizar el proyecto y tratar de mejorar en performance todo lo que podía. No cambié nada de lógica, me limité a cambiar el flujo de datos asíncrono, que era mayormente todo lo que tuviera que ver con leer o escribir en base de datos.

Tras este refactor vi ciertos patrones que quiero remarcar y mostrar cómo podemos darles la vuelta para convertirlos en best practices:

1. Evita async innecesarios

```
export function randomNumber() {  
    return Math.random()  
}  
  
export async function asyncRandomNumber() {  
    return Math.random()  
}
```



@ulisesantana



Estas dos funciones son idénticas a excepción de que la segunda es asíncrona. Sin ningún motivo, pero es asíncrona. Cada vez que usamos `async` en una función, estamos automáticamente haciendo que devuelva una promesa, y eso hay que gestionarlo de más.

Puede parecer una tontería, pero afecta. Hice una pequeña demo para demostrar hasta qué punto esto afecta a nuestro performance. Lo que hace el script es ejecutar cada una de estas funciones por separado un millón de veces.

```
~/projects/talks/nodejs-async-performance/demo  
node async-await/useless-async.mjs  
Executing 1.000.000 times  
sync function: 9.743ms  
async function: 57.74ms
```



@ulisesantana



Como podemos ver, sólo por poner ese `async` hemos hecho que tarde casi seis veces más.

```
describe('This feature should', () => {  
  it('do stuff', async () => {  
    //test...  
  })  
  
  it('do more stuff', async () => {  
    //test...  
  })  
})
```



@ulisesantana



Aplicándolo a la vida real, en una suite de test reducimos un 40% el tiempo que tardaba en ejecutarse sólo quitando los `async` innecesarios que se nos habían quedado después de un refactor. Simplemente quitamos los `async` que teníamos en las arrow functions que ya no nos hacía falta. Haciendo esto pasamos de tardar 2 minutos en tirar la suite de test a poco menos de minuto y medio.

2. Evita los await dentro de los bucles.

Imagina que tienes una tarea que sea: *En base a una lista de IDs tienes que recuperar información de una API*. Por limitaciones técnicas no puedes pasarle a la API la lista de IDs, sino que tienes que hacer una llamada por cada ID. ¿Cuál es la primera idea que se nos viene a la cabeza? Probablemente un bucle, apesta a bucle. La implementación podría ser algo así:

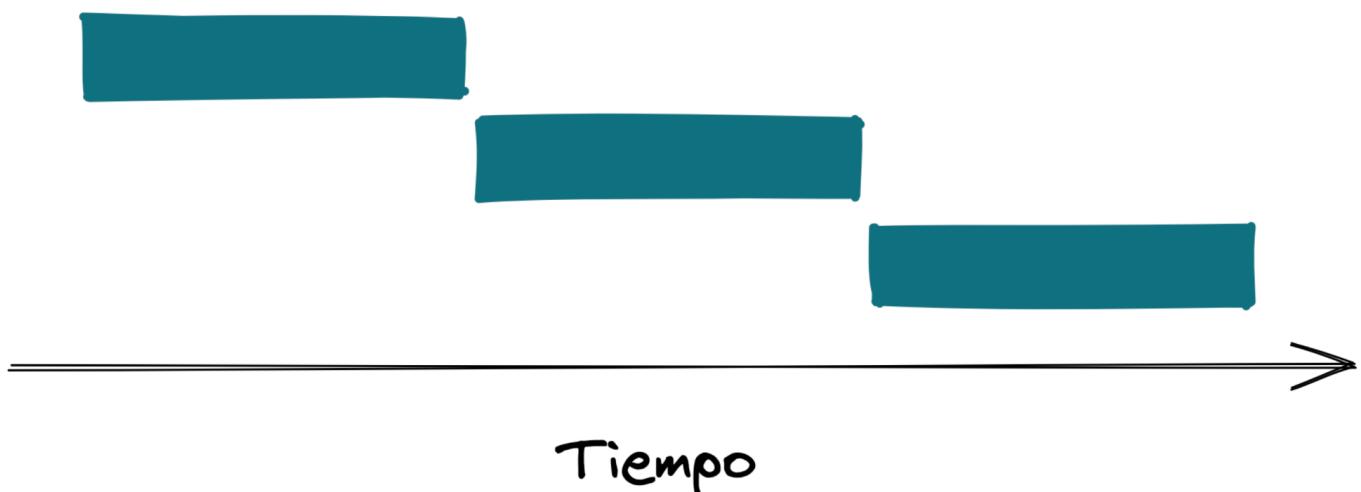
```
async function fetchUserInfo(ids) {  
  const values = []  
  for (const id of ids) {  
    values.push(await fetchUserInfo(id))  
  }  
  return values  
}
```



@ulisesantana



Parece sencillo y además si lo probamos veremos que funciona. Consigue la información de todos los usuarios sin problemas. ¿Sin problemas? Ese await dentro del `for` fuerza a que se termine de completar cada promesa antes de procesar la siguiente.



@ulisesantana



Esto significa que si de media la API tarda en responder 50ms y tenemos 100 IDS que procesar, tardaremos unos 5 segundos en realizar la tarea. No es que sea un drama, pero si implementamos esta otra solución la cosa cambia bastante:

```
function fetchUserInfo(ids) {  
    const values = []  
    for (const id of ids) {  
        values.push(fetchUserInfo(id))  
    }  
    return Promise.all(values)  
}
```



@ulisesantana



Aunque parezca igual aquí hay tres sutiles diferencias:

1. La función ya no es asíncrona, aunque sí que devuelve una promesa.
2. Dentro del for ya no hacemos un await
3. Devolvemos un Promise.all en vez de los valores como hacíamos antes.

La gran diferencia de esta solución es que dentro del bucle no resolvemos las promesas, sino que simplemente las añadimos a nuestro array pendiente de ser resueltas. Cualquier función que devuelva una promesa la devuelve en este estado que hasta que no uses *await* o *.then* no estará *fulfilled* o *rejected*. Podemos verlo como el experimento del gato de Schrodinger, hasta que no abres la caja no sabes si el gato está vivo o muerto. A las promesas les pasa algo parecido, hasta que no las resuelves están en estado *pending* y una vez resueltas pueden estar *fulfilled*, que es cuando se ha resuelto satisfactoriamente y lo que tienes es el valor, o *rejected* que es cuando ha habido algún error y lo que hace es lanzar la excepción que tienes que capturar con un *try/catch* o con el *.catch*.

Volviendo a la solución, vemos que las promesas no se resuelven, sino que se almacenan directamente en estado *Pending*, y la función al devolverlas las resuelve todas *a la vez*. Esto hace que ahora la tarea se haga mucho más rápido, tardando lo que tarde en responder la llamada a la API más lenta.



@ulisesantana



Para ver esto mejor voy a mostrar otra demo en la que usamos casi el mismo código, lo único que cambia es que sustituimos la llamada a la API por una simple espera de 1 milisegundo. Lo que vamos a ver es cuánto tarda cada una de las soluciones ejecutando esta tarea para una lista de 100 IDs y lo va a hacer 1000 veces para que podamos ver si realmente hay una diferencia de performance o no.

```
~/projects/talks/nodejs-async-performance/demo
node best-practices/async-loop.mjs
Async loops for 1000 executions
using Promise.all: 1.291s
using an await inside a for: 1:55.236 (m:ss.mmm)
```



@ulisesantana



Como vemos la diferencia es abismal. Para la misma tarea cuando usamos `Promise.all` tarda poco más de 1 segundo, pero cuando usamos `await` dentro del `for` tarda **casi 2 minutos**. Esta diferencia en un entorno real es crítica y lo peor es que por lo general no nos damos cuenta de que el problema está en esta clase de sitios.

3. Usa Promise.all siempre que puedas

En nuestro día a día desarrollando soluciones de software nos encontramos que en más de una ocasión necesitamos varios recursos de diferentes sitios, ya sean tablas, bases de datos o APIs. Todo esto además tiene una naturaleza asíncrona y necesitamos gestionarla.

```
async function readAllUserInfo(userId) {
  const user = await readUser(userId)
  const contracts = await readContractsForUser(userId)
  const invoices = await readInvoicesForUser(userId)
  return {
    ...user,
    contracts,
    invoices
  }
}
```



@ulisesantana



Aquí vemos un problema parecido al anterior, pero sin bucles. Aunque es menos dramático, es otro de los sitios de donde podemos rascar performance si utilizamos *Promise.all*, ya que como vemos ninguna de las peticiones depende de la otra, por lo que podríamos pedir toda la información a la vez y así reducir el tiempo que necesita la función para realizar la tarea.

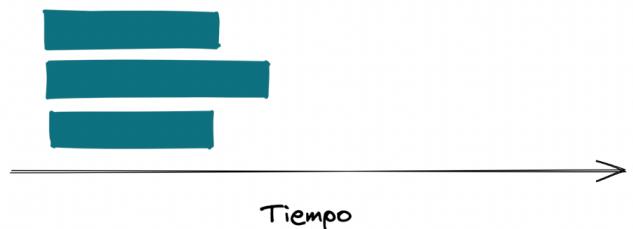
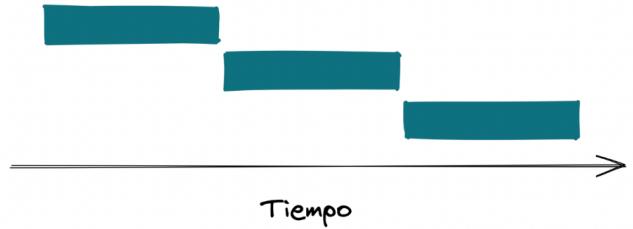
```
async function readAllUserInfo(userId) {
  const [ user, contracts, invoices ] = await Promise.all([
    readUser(userId),
    readContractsForUser(userId),
    readInvoicesForUser(userId)
  ])
  return {
    ...user,
    contracts,
    invoices
  }
}
```



@ulisesantana



Aquí vemos como todas las llamadas se pasan al *Promise.all* y las sacamos por destructuring en el mismo orden en el que se la hemos pasado.



@ulisesantana



Como pasó antes, esto tardará lo que tarde la petición más lenta, en vez de tener que esperar por todas una detrás de otra. El uso de *Promise.all* es bastante sencillo y mejora el tiempo de respuesta de nuestras aplicaciones. Sin embargo, como es tan goloso y vamos a querer usarlo en todos lados hay algo que tenemos que tener en cuenta, tenemos que ser conscientes de cuantas promesas estamos gestionando.

4. Sé consciente de cuantas promesas estás gestionando

```
function fetchUserListInfo(ids) {  
  return Promise.all(ids.map(fetchUserInfo))  
}
```



@ulisesantana



Esta sería otra forma de hacer el *fetchUserListInfo* que hemos visto un par de diapositivas atrás. Tanto esta como la anterior solución tienen un problema, no sabes cuántas promesas vas a tener en el *Promise.all*. En casos en los que no sabes el número de promesas que vas a gestionar o este número es muy alto es recomendable usar la librería *p-map* y limitar la concurrencia. La razón para hacer esto es que si tienes demasiadas promesas puedes acabar haciéndote un ataque de denegación de servicio a ti mismo sin darte cuenta.

En el Proyecto Leñador más que un ataque de denegación de servicio lo que nos preocupaba era ahogar la base de datos. En estos proyectos la práctica habitual era limitar la concurrencia al número de conexiones que teníamos configurado para la base de datos, evitando así ahogarla.

```
import pMap from 'p-map';

function fetchUserInfoList(ids) {
  return pMap(
    ids,
    fetchUserInfo,
    {concurrency: 10}
  )
}
```



@ulisesantana



La diferencia usando *p-map* es que tienes que pasar por separado la lista sobre la que quieras iterar, la función que se ejecutará para cada uno de los elementos de la lista y por último las opciones de *p-map*. En este caso sólo le definimos que queremos que como máximo resuelva 10 promesas a la vez.



@ulisesantana



Por último, tener en cuenta que *p-map* en su versión 5 pasó a ser de tipo ESModules y a menos que tu proyecto esté hecho de esta manera no te va a funcionar. Para poder usarlo con CommonJS necesitas tirar de la versión 4. La realidad es que ambas versiones sólo difieren en si funcionan con ESModules o con CommonJS. Vamos, que si importas cosas en ficheros con *import* nativamente o con *require*.

5. Cachear queries

Esto no es exclusivo de Node.js, pero era algo que no estábamos haciendo y que podía ayudar, ya que había ciertos datos que no cambiaban durante la ejecución y que no venía mal tenerlos cacheados.

Lo que sí es exclusivo de Node.js es cómo cachear esta clase de datos. No cacheas el valor, sino la promesa que te devuelve. Ya después cuando cogenes la promesa cacheada la resuelves y sigues trabajando con normalidad.

Cacheando por valor:

```
function CacheByValue() {  
  let value = undefined  
  
  return {  
    async getNumber() {  
      if (value === undefined) {  
        value = await doSomethingAsync()  
      }  
      return value  
    }  
  }  
}
```

@ulisesantana



Cacheando por promesa:

```
function CacheByPromise() {  
    let value = undefined  
  
    return {  
        getNumber() {  
            if (value === undefined) {  
                value = doSomethingAsync()  
            }  
            return value  
        }  
    }  
}
```

@ulisesantana



Aquí vemos cómo serían las implementaciones de gestión de caché, una por valor y la otra por promesa. Como vemos, la única diferencia es la asignación a la variable la primera vez que se solicita el recurso. En un caso usa el await, en el otro no.

Tengo otra demo hecho a correr estos dos trozos de código cien millones de veces para ver si realmente hay diferencia. La realidad es que la hay y es recomendable cachear la promesa en vez del valor si queremos seguir rascando performance.

```
~/projects/talks/nodejs-async-performance/demo  
node best-practices/cached-promises.mjs  
Caching promises for 100.000.000 executions  
by promise 100000000: 3.895s  
by value 100000000: 5.387s
```



@ulisesantana



6. Haz caso de los warnings

No sé si a alguien más le pasa que la mayoría del tiempo ignoras los warnings y sólo le das importancia cuando son errores. En la consola al ejecutar el proceso me salía esto:

```
(node) warning: possible EventEmitter memory leak detected.  
11 listeners added.  
Use emitter.setMaxListeners() to increase limit.
```



@ulisesantana



El mensaje nos dice que Node.js ha detectado un posible *memory leak* debido al *Event Emitter* porque se están añadiendo más listeners de los recomendados. Te ofrece la opción de incrementar el límite para en caso de que tengas claro lo que estás haciendo y lo necesites.

Yo pensaba “*Meh, es un warning*”. En una primera instancia simplemente hice lo que me decía y aumenté los listeners sin darle mayor importancia. Sin embargo, el proceso todavía era demasiado lento así que investigué el warning.

El problema era que había event handlers que se estaban creando continuamente con cada conexión que se solicitaba al pool de conexiones de la base de datos, pero que no se estaban eliminando, siendo el causante del memory leak. Tras implementar el fix en el que cada vez que se devuelve una conexión al pool se limpian los event handlers asociados a la conexión vimos una mejoría en la performance, tardando 4 veces menos de lo que tardaba antes.

Prototipo original:

~7 minutos

Proyecto Leñador:

4 minutos y 52 segundos



@ulisesantana



Con todas estas mejoras en el Proyecto Leñador lo volvimos a ejecutar con el mismo set de datos y en esta ocasión tardó **04:52**. Era incluso más rápido que el prototipo original que tardaba 7 minutos. En ese momento mi yo interior era algo así:



@ulisesantana



No recuerdo si era de día, de noche, ni que hora era. Sólo recuerdo ese sentimiento de **FUCK YEAH**, ese paso de la ansiedad a la paz.

Prototipo original:

41 minutos y 29 segundos

Proyecto Leñador:

31 minutos y 56 segundos



@ulisesantana



Y ya por estar completamente seguros de que había una mejora, ejecutamos una prueba con un set de datos semejante al que se enfrentaba en producción día a día, que eran millones de registros en la base de datos. En este caso el resultado quedó claro:

- 20%



@ulisesantana



Aparte de haber mejorado la sostenibilidad del proyecto, habíamos mejorado la performance haciendo que el proceso fuera más de un 20% más rápido. Y todo esto llegando al deadline. Al final todo salió bien, pero sin duda aprendí un par de cosas de esta experiencia:

Aprendizaje

1. La asincronía en JS está más incomprendida de lo que pensaba
2. Forma a tu equipo
3. La responsabilidad debe ser compartida
4. Sé prescindible



@ulisesantana



- **La asincronía en JS está más incomprendida de lo que pensaba.** La gestión de asincronía en JavaScript es algo que no todo el mundo tiene interiorizado. Todo el mundo usa promesas y el async/await y empezar a trabajar con JavaScript o TypeScript no es tan complicado, lo que sí es más complicado es saber cómo gestionar la asincronía en JavaScript. Aclarar, que la asincronía en sí es algo que no es fácil de comprender, ya que esto pasa en otros lenguajes también.
- **Forma a tu equipo,** comparte el conocimiento. Busca tiempo para poco a poco ir formándolo. En mi experiencia el mob programming ayuda bastante, pero no es suficiente. Algunos conceptos necesitas interiorizarlos y para eso lo mejor es hacer katas o tener formaciones con objetivos concretos. Al principio es bastante duro preparar esta clase de formaciones, pero a medida que las tengas podrás reusarlas a medida que entren personas nuevas o si cambias de equipo puedes formar a ese nuevo equipo.
- **La responsabilidad debe ser compartida.** Es bastante típico que las personas con más experiencia del equipo se echen las cosas a la espalda. La responsabilidad debe ser compartida, tanto tecnológica como a nivel de diseño de software o metodologías. La idea es que los miembros del equipo aprendan unos de otros sin importar el nivel de experiencia que tengan. El intercambio de ideas desde distintos puntos de vista puede enriquecer mucho al equipo.
- **Sé prescindible.** Si eres prescindible no serás el cuello de botella. En esta historia yo fui un cuello de botella por conocimiento y por dominio. Es algo de lo que me arrepiento, pero también algo de lo que aprendí. Ojo, digo prescindible que no innecesario. Con esto quiero decir que no seas crítico por conocimiento, ya sea de dominio, procesos o tecnología. Trata siempre de compartir tu conocimiento con el equipo y documentarlo. Todo lo que tiene un inicio tiene un fin, por lo que algún momento dejarás de estar en el equipo en el que estás ahora. Cuando eso pase lo importante es que no te lleves conocimiento contigo, sino que lo hayas dejado en el equipo en forma de documentación. De esta manera serás prescindible.

Y esta ha sido la historia y el aprendizaje de cómo pasé un proceso en Node.js de 5 horas a 5 minutos. Resumiendo estos son mis 6 consejos cuando gestionas asincronía en JavaScript:

Cómo gestionar asincronía en JavaScript

1. Evita los async innecesarios
2. Evita los await dentro de los bucles
3. Usa Promise.all siempre que puedas
4. Sé consciente de cuantas promesas estás gestionando
5. Cachea la promesa en vez del valor que resuelve
6. No ignores los warnings



@ulisesantana



Bonus tips

Tengo un par de cosas más en el tintero que no son producto de esta experiencia con el *Proyecto Leñador*, sino del día a día durante los últimos años. Y como tengo tiempo quiero compartirlas.

Bonus tips

1. No mezcles tipos de asincronía
2. Descubre los async generators
3. Investiga sobre el Event Loop



@ulisesantana



1. **No mezcles tipos de asincronía:** Aparte de ser más difícil de leer, también acaba afectando al performance porque la mayoría de las veces lo que hacemos es complicar el comportamiento asíncrono.
2. **Async generators, ese gran desconocido:** En JavaScript existe una cosa llamada *generators*, de la cual llegó hace un par de años su versión asíncrona. Los *async generators* donde brillan es al gestionar *streams*. Sin embargo, hay otros casos de uso que también son muy útiles. Lucciano Mammino dió una [charla sobre ellos en la NodeCONF EU en 2019](#).
3. **Investiga sobre el Event Loop:** Entender cómo funciona JavaScript por debajo te ayuda a entender por qué a veces las cosas no pasan como esperamos. En la documentación de Node.js hay un apartado de guías en la cual hay una llamada [The Node.js Event Loop, Timers, and process.nextTick\(\)](#). Es un documento que tardaron 6 meses en escribir y que explica los diferentes procesos y el orden en el que se ejecutan en cada vuelta del Event Loop en Node.js.

También dejo por aquí [este artículo](#) de James Snell sobre gestión de promesas en Node.js.