Llamadas asíncronas (Parte I)

La asincronía

Competencias

- Reconocer el concepto de asincronía en aplicaciones multiprocesos para resolver un problema
- Identificar el uso de los callbacks para resolver un problema planteado

Introducción

La asincronía es un comportamiento que está presente en el mundo del software web en todo momento, por el simple hecho de tener operaciones ejecutándose de forma simultánea.

El ejemplo más común para representar la asincronía en una aplicación es cuando se necesita consumir datos externos a la aplicación en donde estemos desarrollando, esto requiere de tiempo de procesamiento y debemos manejarlo con herramientas asíncronas.

En este primer capítulo comprenderás que es la asincronía y a través de pequeños ejemplos en donde ocuparemos las funciones "callbacks" adquirirás las competencias necesarias que te permitirán desarrollar aplicaciones o sistemas de manera eficiente utilizando el entorno de Node.

¿Qué es la asincronía?

Para entender el concepto de asincronía, debemos primeramente recurrir a su opuesto, es decir, la sincronía:

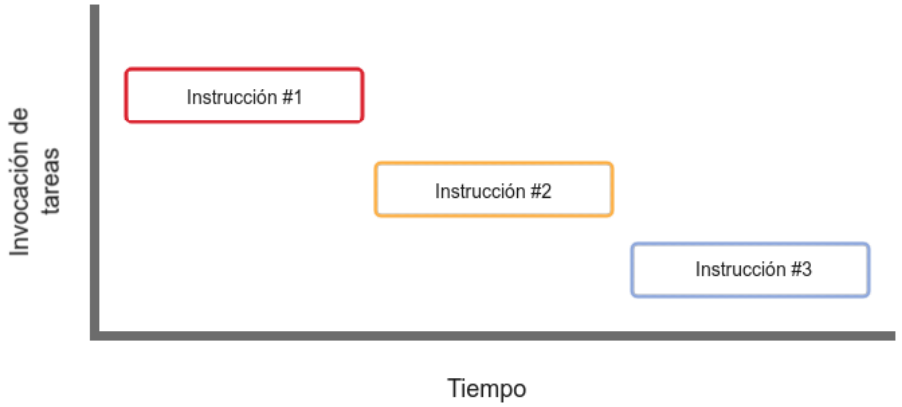

Imagen 1. Sincronía en un contexto de ejecución.
Fuente: Desafío Latam

En la Imagen 1, se muestra el concepto de sincronía en un contexto de ejecución de instrucciones. La sincronía se refiere a la coincidencia con respecto al orden y tiempo de invocación de un conjunto de instrucciones, comprendiendo que una llamada es una instrucción que el motor Node debe resolver. Entonces, una instrucción posee la característica de sincronía, sí por ejemplo, debemos ejecutar la operación de inmediato para permitir la continuación de ejecución del hilo de procesos. Por el contrario, se consideraría asíncrona sí la llamada puede ser diferida, anclada o ejecutada en paralelo y de forma simultánea, debido a que no depende de un cierto orden en la secuencia de ejecución.

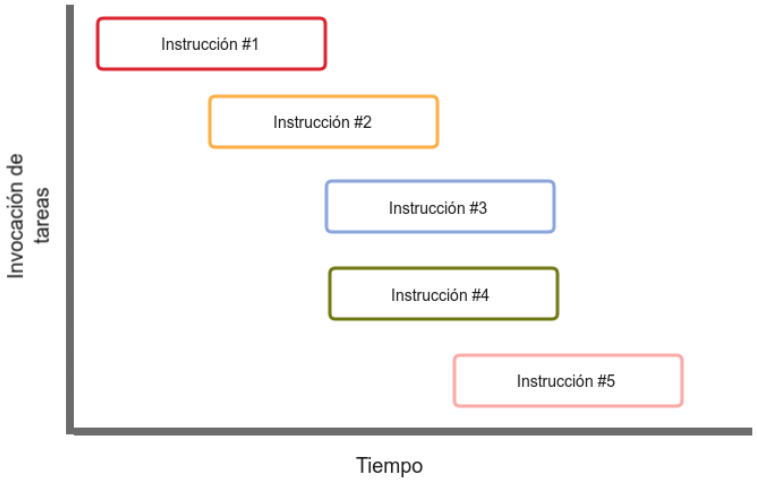

Imagen 2. Asincronía en un contexto de ejecución.
Fuente: Desafío Latam

Según la RAE, define asincronismo como la falta de coincidencia de dos o más hechos en un espacio tiempo.

En la Imagen 2 se muestra el concepto de asincronía en un contexto de ejecución de instrucciones o procesos. Veámoslo a detalle con la siguiente lista:

- La instrucción #1 no depende directamente de la instrucción #2.
- Las instrucciones #3 y #4 son paralelas y simultáneas.
- Con respecto a la última instrucción no podemos inferir a ciencia cierta si fue invocada producto de la #2, #3 o #4 instrucción.

Para que este comportamiento sea posible en una aplicación con Node, disponemos del concepto de callbacks, el cual recordemos que se refiere a la ejecución de una función como respuesta de otra función.

Veamos un ejemplo básico de un código con callbacks donde enviaremos como argumento una función a otra función, el objetivo será imprimir por consola el nacimiento de un pollito desde su cascarón, su reciente nacimiento y su etapa adulta. La gracia en este ejemplo es que la etapa del medio (nacimiento del pollito) sea ejecutado por medio de una función pasada como argumento a la función "principal", que en este caso llamaremos piopio. Observa con detención el siguiente código:

```
// Definimos la función que implementará el callback
const piopio= function (callback) {
 // Añadimos una impresion
 console.log("🥚")
 // Ejecutamos nuestro callback
 callback()
 // Añadimos otra impresión
 console.log("🐔")
}

// Utilizamos nuestra función
piopio(function () {
   console.log("🐤")
})
```

Si ejecutamos el archivo en un entorno de Node, obtendremos un resultado similar al presentado a continuación:

En relación con lo anterior, se puede resumir que podemos enviar una función como argumento de otra función, y ésta es necesaria para la lógica de la función principal. Sí comprendemos los conceptos de asincronía y callbacks podremos aplicarlos en la construcción de cualquier tipo de programa en el entorno de Node.

Para darte un ejemplo un poco más directo al día a día, la asincronía está presente en la idea de consultar a un servidor remoto, implicando directamente manejar sucesos sin una sincronía. ¿Por qué? Porque al necesitar consumir datos de una aplicación alojada en otro servidor, la información la queremos usar en procesos consecuentes y entramos inevitablemente en la necesidad de esperar que la data llegue para poder proceder con nuestras instrucciones, además de programar la secuencia de sucesos en relación al problema que tengamos planteado.

Problemas comunes con callbacks

Entendiendo que un callback es una función que se ejecutará después de que otra función se haya ejecutado, ¿Qué pasaría si dentro de un callback tenemos otro callback, y así consecutivamente por varios niveles? El código es difícil de entender y mantener, esto se conoce como "El infierno de los callbacks".

El infierno de los callbacks

El infierno de los callbacks o callback hell se refiere a la utilización de múltiples funciones asíncronas anidadas, lo que puede dificultar la comprensión y el mantenimiento del código.

En la siguiente imagen te muestro un ejemplo de este caso.

```
// Callback Hell


a(function (resultsFromA) {
    b(resultsFromA, function (resultsFromB) {
        c(resultsFromB, function (resultsFromC) {
            d(resultsFromC, function (resultsFromD) {
                e(resultsFromD, function (resultsFromE) {
                    f(resultsFromE, function (resultsFromF) {
                        console.log(resultsFromF);
                    })
                })
            })
        })
    })
});

```

Imagen 3. Sincronía en un contexto de ejecución.
Fuente: https://medium.com

¿Ahora entiendes por qué el nombre? Claramente este código necesita que leamos detenidamente línea por línea lo que está sucediendo, porque no está claro de primera (a pesar de ser un ejemplo básico).

Hacer muchos callbacks de forma anidada hace que nuestro código crezca en indentación, lo hace menos legible y mantenible. A ciencia cierta no existe una solución definitiva al infierno de los callbacks (en caso de querer usarlos), sin embargo hay un conjunto de buenas prácticas a seguir y metodologías a implementar si queremos evitar caer en el infierno de los callbacks, algunas de estas son las siguientes:

- Mantén todo en el máximo orden posible.
- Define los alcances de las variables y funciones.
- Manejar todas las posibles excepciones de tu programa.
- Considerar utilizar algún patrón para manejar mutaciones de estados o variables.
- Considerar implementar promesas o funciones asíncronas

La última recomendación es tal vez la más importante y es que ES6 nos trajo las herramientas necesarias para solucionarlo. Sin embargo, no se puede decir que los callbacks están en proceso de extinción, siguen siendo una herramienta importante a considerar, al día de hoy siguen saliendo documentaciones oficiales de tecnologías de reconocimiento que los utilizan, pero no de una manera abusiva.

Ejercicio propuesto (1)

Completa las siguientes frases y debate con tus compañeros las soluciones.:

- Una función que es ejecutada como respuesta de otra se considera un ____
- El concepto _____ se refiere a una secuencia de varios callbacks anidados.

Resolviendo un caso asíncrono

Competencias

- Identificar el uso de los callbacks, async await y promesas en el planteamiento de un problema asíncrono
- Construir un programa que utilice async/await y promesas para resolver un problema de asincronía acorde al entorno Node

Introducción

En este capítulo aprenderás a través del uso de diferentes herramientas de JavaScript cómo abordar problemas de asincronía en Node. Estas herramientas son callbacks, promesas y async/await en donde cada una representan escenarios diferentes y te permitirá crear programas más sencillos de mantener y menos impredecibles al momento de ser ejecutados y operados.

Además, verás cómo las promesas y async/await aparecieron con ES6 a solucionar este caso particular y ofrecerte una forma más cómoda de programar y de leer el código de tus aplicaciones.

Los callbacks en Node

No existe una diferencia entre los callbacks aplicados en el frontend y aplicados en el backend, se trata del mismo lenguaje de programación por lo que no tendrás que aprender una nueva forma de escribir callbacks, y es que ya lo has hecho en sesiones anteriores. ¿No sabes cuándo? Déjame refrescarte la memoria con el siguiente código:

```
fs.writeFile('message.txt', data, (err) => {
  if (err) throw err;
  console.log('El archivo ha sido creado con éxito!');
});
```

El callback cuando usamos los métodos asíncronos de File System corresponden al tercer parámetro, esto quiere decir que esa función se va a ejecutar como respuesta del intento de creación de (en este caso) el archivo “message.txt”, recibiendo como parámetro el posible error de la instrucción.

Este es un ejemplo de uso local, no obstante los callbacks tienen mucho más sentido requerirse cuando necesitamos consumir datos de un servidor remoto. Para esto se recomienda usar las promesas, en la cual también se ocupan los callbacks pero de una forma mucho más clara.

Las Promesas en Node

Al igual que en los callbacks, no hay diferencia del uso de promesas en el frontend y backend cuando hablamos de desarrollo con JavaScript, no obstante veamos un repaso y un ejemplo.

Hagamos un pequeño repaso, recordando que para declarar una promesa tenemos que crear un nuevo objeto de tipo Promise utilizando el operador “new”. El método constructor o la función “Promise” recibe un callback de parámetro, el cual define dos parámetros: “resolver” y “rechazar”. Tal como se puede observar en el siguiente fragmento de código:

```
// Una promesa en una constante ...
const promesa = new Promise((resolver, rechazar) => {
   // Contexto de la promesa ...
});
```

Dentro del contexto de la promesa, tenemos que definir la lógica que deberá determinar si la promesa fue resuelta o rechazada. A modo de ejemplo en el siguiente código te muestro el uso de una promesa con una temática similar a una lotería, es decir, que de forma aleatoria

se generará un número y si el resultado cumple una condición se enviará un mensaje de éxito, de lo contrario se envía un mensaje de fracaso.

```
// Definición del callback de la promesa:
const callback = (resolver, rechazar) => {
 // Con un 50% de probabilidad:
 if (Math.random() >= 0.5) {
   // Podrá fallar.
   rechazar("El número resultante fue menor a 0.5 :/")
} else {
   // De lo contrario tendrá éxito dentro de 5 segundo.
   setTimeout(function(){
     resolver("El número resultante fue mayor o igual a 0.5 B) ");
  }, 5000);
}
}

const promesa = new Promise(callback);
```

En este caso, se utilizó la función random del objeto Math de JavaScript, para establecer una probabilidad del 50% de fallo en la ejecución de la promesa, invocando el parámetro rechazar con un mensaje de error como argumento. Si tenemos suerte, luego de 5 segundos nuestra promesa nos debería retornar el mensaje "¡Éxito!" mediante la invocación del parámetro resolver(esto es un callback) aplicando un mensaje como argumento.

Para implementar una promesa debemos utilizar el método "then", el cual recordemos inicia una cadena de promesas, permitiéndonos escribir un código más ordenado y particionado; si quieres saber más sobre promesas y cómo encadenarlas puedes revisar la documentación oficial de Promesas. Además tenemos el método "catch" de la promesa que nos devolvería el error en caso de existir.
Ambos métodos deben ser implementados definiendo al menos un parámetro correspondiente a los parámetros pasados en los métodos resolver y rechazar.

Un ejemplo de ello sería el siguiente código:

```
promesa.then((mensaje) => {
 console.log("¡Sí! La promesa se resolvió sin problema y el mensaje
resultante es:");
  console.log(mensaje)
}).catch((error) => {
 console.log("Bah... Algo salio mal :(, el error es el siguiente:")
 console.log(error)
});
```

En caso que la promesa tenga éxito (considerando la probabilidad de fallo), esto nos podría imprimir en la consola el siguiente mensaje:

```
¡Sí! La promesa se resolvió sin problema y el mensaje resultante es:
El número resultante fue mayor o igual a 0.5 B)
```

Y en caso de fracasar la promesa, obtendremos el siguiente mensaje:

```
Bah... Algo salio mal :(, el error es el siguiente:
El número resultante fue menor a 0.5 :/
```

Sobre las promesas podemos realizar las siguientes afirmaciones:

- Requiere el uso de callbacks.
- Permite secuenciar la ejecución de contextos utilizando then.
- Habilita el manejo de errores mediante el uso de catch.
- Crea un contexto intermediario para manejar operaciones asíncronas.

Promesas y callbacks son conceptos diferentes. El primero de ellos fue definido recientemente en el estándar ES6 y el segundo es un método de programación de algoritmos asíncronos. El concepto promesa aparece como otro método para implementar algoritmos asíncronos pero más confiables, legibles y mantenibles.

Async/Await en Node

En JavaScript, a veces necesitamos voluntariamente especificar que una función se ejecute de forma asíncrona, además de especificar dentro de la misma, cuál será la instrucción que necesitamos que se resuelva indispensablemente antes de seguir con las demás instrucciones, para esto al igual que en el desarrollo frontend tenemos “async” y “await”.

Los callbacks, las promesas y async/await están altamente relacionadas a nivel avanzado, las utilizamos en todo momento. Te podrás estar preguntando ¿En donde se relacionan? para comprender esto hay que entender que la palabra reservada “async” se debe escribir antes de una función y esto hará que en la lectura, la función se interprete como una promesa y como en toda promesa podemos ocupar el método then para concatenar funcionalidades.

Diferencia de funciones con Async y sin Async

Veámoslo con un ejemplo. En el siguiente código te muestro un caso sin async en donde la temática es una persona que quiere pedalear una bicicleta:

```
function montarse(){
  console.log('Montandome en la bicicleta')
}

function pedalear(){
  console.log('Pedaleando')
}

montarse()
pedalear()
```

Al ejecutarse este código por supuesto recibiremos el siguiente mensaje

```
"Montandome en la bicicleta"
"Pedaleando"
```

Nada nuevo por acá. Ahora si entendemos que no tendría sentido ejecutar la función "pedalear" antes que "montarse" pudiéramos querer que si o si se ejecuten en el orden correcto, para esto podríamos asignarle a la función "montarse" la palabra reservada async y a la ejecución concatenarle la función "pedalear". Te lo muestro en el siguiente código:

```
async function montarse(){
  console.log('Montandome en la bicicleta')

}

function pedalear(){
  console.log('Pedaleando')
}

montarse().then( () => {
  pedalear()
})
```

Y esto nos dará como resultado el siguiente mensaje:

```
"Montandome en la bicicleta"
"Pedaleando"
```

Como puedes notar, es lo mismo, el resultado no ha cambiado pero funcionalmente hemos dejado de ejecutar las funciones en instrucciones diferentes y pasamos a concatenar sus usos de una forma más lógica.

Te podrás estar preguntando ¿En dónde está la magia? Lo sé, hasta ahora podría parecer que no hay ninguna importancia ni necesidad en usar async, y es que esto solo fue para enseñarte que podemos concatenar funciones "convirtiéndolas" en promesas, y entendiendo que las promesas pueden ocupar los "then" de forma infinita generando una cadena de promesas, pudiésemos tener una estructura más entendible, porque quien lea el código de primera entenderá la relación secuencial que hay entre diferentes funciones.

¿Y qué pasó con el await? He aquí el uso más interesante. El await se declara antes de un proceso que esté condicionado a un caso de éxito o fracaso y ¿A qué suena esto? Exactamente, a una promesa.

Veámoslo con otro ejemplo pero en este caso usamos la librería axios puesto que el "await" tiene sentido usarlo solo cuando hay un proceso que necesita tomarse un tiempo para resolverse.

Ejercicio guiado: El último sismo

Desarrollar una aplicación que al ser ejecutada devuelva la fecha del último sismo en Chile y su magnitud. Para esto utilizaremos una de las APIs de https://gael.cloud/ en su endpoint https://api.gael.cloud/general/public/sismos:

Inicia un proyecto con el comando **"npm init -y"** y luego instala axios con el comando **"npm i axios"**, posteriormente sigue los siguientes pasos:

- ***Paso 1:*** Importar axios en una constante.
- ***Paso 2:*** Crear dos variables llamadas "fecha" y "magnitud".
- ***Paso 3:*** Crear una función asíncrona llamada "getData".

- ***Paso 4:*** Crear una variable que extraiga el objeto "data" de una consulta con axios al siguiente endpoint, https://api.gael.cloud/general/public/sismos

    - La consulta de la API debe tener por delante la palabra reservada "await".
    - La variable "data" terminará siendo un arreglo con los últimos sismos registrados en la API.

- ***Paso 5:*** Realizar las siguientes asignaciones:

    - Asignar a la variable fecha, el valor de la propiedad "Fecha" del primer objeto del arreglo "data".
    - Asignar a la variable magnitud, el valor de la propiedad "Magnitud" del primer objeto del arreglo "data".

- ***Paso 6:*** Crear una función llamada "imprimir", que imprima por consola el siguiente mensaje

```
`El último sismo fue el ${fecha} y tuvo una magnitud de ${magnitud}`
```

- ***Paso 7:*** Ejecutar la función asíncrona "getData" y en su then ejecuta la función "imprimir".

```
// Paso 1
const axios = require('axios')
// Paso 2
let fecha
let magnitud
// Paso 3
async function getData() {
 // Paso 4
 let { data } = await
axios.get('https://api.gael.cl/general/public/sismos')
 // Paso 5
 fecha = data[0].Fecha
 magnitud = data[0].Magnitud
}
// Paso 6
function imprimir() {
 console.log(`El último sismo fue el ${fecha} y tuvo una magnitud de
${magnitud}`)
}
// Paso 7
getData().then(() => {
```

```
 imprimir()
})
```

Al ser ejecutada esta aplicación obtendrás lo que se muestra en la siguiente imagen:

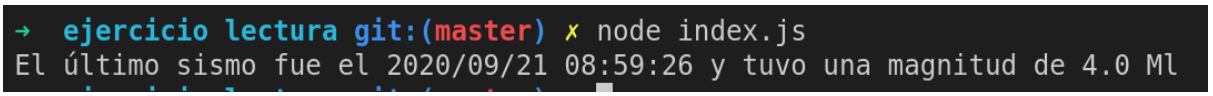

Imagen 4. Mensaje por consola del último sismo obtenido por una función asíncrona.
Fuente: Desafío Latam

¡Muy bien! Hemos obtenido sin problema lo que buscábamos, a modo de refuerzo prueba ejecutar la aplicación de nuevo, pero en esta ocasión quita la palabra reservada "await" y deberás recibir lo que verás en la imagen 5.

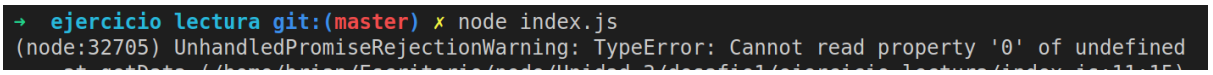

Imagen 5. Mensaje por consola indicando que no se puede leer la propiedad 0 de undefined.
Fuente: Desafío Latam

¿Por qué sucede esto? Correcto, porque al no declarar a la función asíncrona, debe esperar que axios termine de realizar la consulta, el programa se seguirá leyendo y se topará con que se está usando un dato que realmente aún no está definido puesto que la data aún no ha llegado.

Como dato extra y para cuando realices debugging a tu código, es importante que sepas que al declarar una función asíncrona de esta manera hará que la función deje de ser una función síncrona simple y pasa a convertirse en un objeto tipo "AsyncFunction".

Ejercicio propuesto (2)

Basado en el ejercicio de los sismos, desarrolla una aplicación que utilice una función asíncrona que consulte un usuario random en el siguiente endpoint https://randomuser.me/api/. Al usar el método "then" de esta función llamar a otra función que imprima el nombre y la fecha de nacimiento del usuario recibido.

Usando la asincronía en Node

Competencia

- Construir una aplicación que ejecute y capture el resultado de varias promesas a través del Promise.all()

Introducción

En este capítulo, veremos cómo usar las herramientas asincrónicas aprendidas en el capítulo anterior, en una aplicación donde se necesita consumir dos APIs diferentes, cuya data se utilice con un mismo fin.

Con la aplicación que desarrollaremos estarás listo para abordar situaciones similares y a empezar a hacer un buen uso de los callbacks, las promesas y el async/await. Destacando principalmente el uso mixto de estas 3 herramientas obteniendo como resultado un código limpio, legible, más fácil de mantener y con buenas prácticas.

Ejecutar varias promesas en paralelo

Podría suceder la necesidad de ejecutar 2 promesas en paralelo sacándole provecho a los conocimientos de sincronía y asincronía que aprendiste en el primer capítulo, esto es posible con el uso de "Promise.all()", el cual ejecutará simultáneamente todas las promesas que le especifiquemos dentro de un arreglo, nos entregará en caso de resolver exitosamente todas las promesas, el resultado de cada una y en caso de que al menos 1 falle, recibir el error correspondiente.

Ejercicio guiado: Ejecutando promesas en paralelo

Desarrollar una aplicación que contenga dos promesas, ambas devolverán un número y se ejecutarán con el Promise.all() para finalmente sumar los resultados y enviar el total por consola. Realiza los siguientes pasos para la solución de este ejercicio:

- ***Paso 1:*** Crear una función que retorne una promesa que a su vez retorna el número 1.
- ***Paso 2:*** Crear una función que retorne una promesa que a su vez retorna el número 2.
- ***Paso 3:*** Usar Promise.all para ejecutar ambas promesas creadas pasadas como dentro de un arreglo.
- ***Paso 4:*** Imprimir por consola el resultado del Promise.all.
- ***Paso 5:*** Guardar en una variable la suma de ambos resultados usando el método reduce.
- ***Paso 6:*** Imprimir por consola el resultado de la suma.

```
// Paso 1
function promesa1() {
 return new Promise((res, rej) => {
   res(1)
 })
}
// Paso 2
function promesa2() {
 return new Promise((res, rej) => {
   res(2)
 })
}

// Paso 3
Promise.all([promesa1(), promesa2()]).then((resultado) => {
 // Paso 4
 console.log('Resultado de las promesas: ', resultado)
 // Paso 5
 let suma = resultado.reduce((total, num) => total + num)
 // Paso 6
 console.log('La suma de ambos resultados es: ' + suma)
})
```

Al ejecutar esta aplicación obtendrás lo que se muestra en la siguiente imagen:

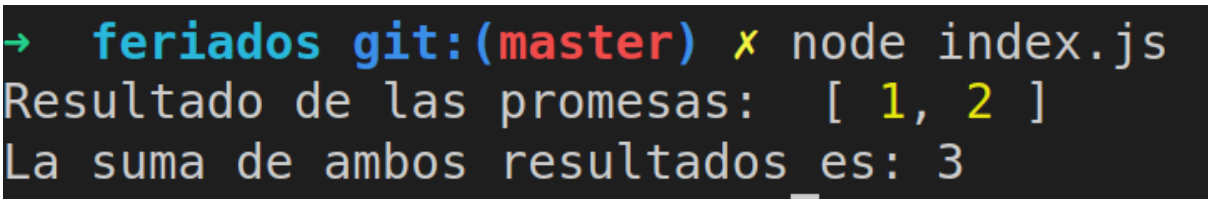

Imagen 6. Mensaje por consola indicando el resultado de ambas promesas.
Fuente: Desafío Latam

¡Excelente! Ya has aprendido cómo ejecutar dos promesas y poder manipular sus resultados.

Ejercicio propuesto (3)

Basado en el ejercicio de las promesas que devuelven un número, desarrolla una aplicación con dos promesas que devuelvan tu nombre y tu apellido y usando el Promise.all imprime la concatenación del resultado de ambas promesas.

Simplificando código con Async/Await

Como ya sabes, asignarle la palabra reservada "async" a una función, esto la convierte en una promesa, así que podríamos simplificar nuestro programa recién creado aplicando los cambios que te muestro en los siguientes pasos:

- ***Paso 1:*** Sustituir la primera promesa por una función asíncrona que retorne el número 1.

- ***Paso 2:*** Sustituir la primera promesa por una función asíncrona que retorne el número 2.

```
// Paso 1
async function promesa1() {
 return 1
}
// Paso 2
async function promesa2() {
 return 2
}

Promise.all([promesa1(), promesa2()]).then((resultado) => {
 console.log('Resultado de las promesas(funciones asincronas): ',
resultado)
 let suma = resultado.reduce((total, num) => total + num)
 console.log('La suma de ambos resultados es: ' + suma)
})
```

Ahora al ejecutar esto veremos lo que te muestro en la siguiente imagen:

```
→  feriados git:(master) ✗ node index.js
Resultado de las promesas(funciones asincronas):  [ 1, 2 ]
La suma de ambos resultados es: 3
```

Imagen 7. Mensaje por consola indicando el resultado de ambas promesas.
Fuente: Desafío Latam

Como puedes apreciar obtenemos el mismo resultado, de esta manera estamos mezclando las funciones asíncronas con las herramientas del objeto Promise para el tratamiento de promesas.

Ejercicio propuesto (4)

Basado en el ejercicio propuesto 3, sustituye las promesas por funciones asíncronas y consigue el mismo resultado.

La asincronía y el consumo de APIs

Competencia

- Construir una aplicación en Node que utilice dos APIs para resolver un problema planteado

Introducción

La asincronía y el consumo de APIs están altamente relacionados en el desarrollo de aplicaciones, ahora que sabemos usar herramientas asincrónicas, en este capítulo aprenderemos a desarrollar una aplicación que solucione un problema planteado, donde necesitaremos consumir la data de 2 APIs, una API nos servirá para conocer las fechas feriadas de chile y la otra que nos entregará un usuario random en cada consulta.

El objetivo de este capítulo práctico será ejecutar estas 2 consultas de forma simultánea y procesar sus respuestas para descubrir si algún usuario aleatorio de la api randomuser cumple años en alguna de las fechas feriadas obtenida de la api feriadosapp.
Con esta práctica aplicarán los conocimientos adquiridos en capítulos previos y estarás más listo para enfrentarte a problemas similares que cotidianamente resolvemos en el mundo laboral.

Usando Promise.all y Async/Await en un problema planteado

Lo siguiente será hacer uso de lo aprendido para resolver el siguiente ejercicio.

Ejercicio guiado: Carrete en mi cumpleaños

Crear una aplicación que dada una fecha de nacimiento evalúe si el día del cumpleaños coincide con un día feriado en Chile. Para esto usaremos las APIs de https://randomuser.me/ y https://www.feriadosapp.com/ en los siguientes endpoints:

- https://randomuser.me/api/
- https://www.feriadosapp.com/api/holidays.json

Realiza los siguientes pasos para la solución de este ejercicio:

- ***Paso 1:*** Importar axios en una constante.

- ***Paso 2:*** Crear una función asíncrona que espere y devuelva un usuario random.

- ***Paso 3:*** Crear una función asíncrona que espere y devuelva los días feriados.

- ***Paso 4:*** Ejecutar ambas funciones asincrónicas con el Promise.all.

- ***Paso 5:*** Guardar el usuario obtenido de la función asíncrona en una variable y genera las siguientes variables
    - Nombre: Concatena el primer nombre y el apellido que se encuentran en el objeto “name” del usuario.
    - Cumpleaños: Necesitarás la fecha de nacimiento del usuario que consigues en el objeto “dob”.
        - Esta variable debe ser la concatenación del mes y el día recordando que al usar getMonth obtenemos los números del 0 al 11 representando los meses, por lo que deberás sumarle 1 para tomar correctamente el número del mes
        - Para la fecha debes considerar que al usar el getDate obtenemos un número sin 0 por delante cuando la fecha es menor a 10, por lo que debes concatenarle un 0. Estas consideraciones las debes tomar para poder hacer la lógica de comparación con el formato de fecha que nos devuelve la API de feriados

- ***Paso 6:*** Recorrer el arreglo de fechas feriadas y guardar en una variable la coincidencia de la propiedad “date” y el cumpleaños del usuario.

- ***Paso 7:*** Crear un operador ternario que evalúe si la variable anterior es "true" devolviendo un mensaje de éxito o fracaso según corresponda.

```
// Paso 1
const axios = require('axios')

// Paso 2
async function getUser() {
 const { data } = await axios.get('https://randomuser.me/api/')
 const user = data.results[0]
 return user
}
// Paso 3
async function getHolidays() {
 const { data } = await
axios.get('https://www.feriadosapp.com/api/holidays.json')
 return data.data
}

// Paso 4
Promise.all([getUser(), getHolidays()]).then((resultado) => {
 // Paso 5
 const user = resultado[0]
 const nombre = `${user.name.first} ${user.name.last}`
 const nacimiento = new Date(user.dob.date)
 const cumpleaños = `${nacimiento.getMonth() + 1}-${
   nacimiento.getDate() < 10 ? '0' + nacimiento.getDate() :
nacimiento.getDate()
 }`

 // Paso 6
 const feriados = resultado[1]
 const carrete = feriados.find((f) => f.date.includes(cumpleaños))

 // Paso 7
 carrete
   ? console.log(`
   'Prepárense todos! porque ${nombre} estará de cumpleaños en el
feriado ${carrete.title}'
 `)
   : console.log(`${nombre} no cumple años en ningún día feriado :/`)
})
```

Ahora si ejecutas esta aplicación y el usuario obtenido con la API cumple años en un día feriado en Chile recibirás un mensaje como el que se muestra en la siguiente imagen:

```
feriados git:(master) ✗ node index.js

'Preparense todos! porque Iiris Pelto estará de cumpleaños en el feriado Inmaculada Concepción'
```

Imagen 8. Mensaje por consola indicando que el usuario random cumple años un día feriado
Fuente: Desafío Latam

Cabe destacar que las probabilidades que ejecutes la aplicación y el usuario recibido por randomuser cumpla años algún día feriado en Chile es considerablemente pequeña, por lo que te recomiendo que si quieres probar el código generes un String con tu fecha de cumpleaños para simular un caso de éxito.

Lo más probable es que no suceda la coincidencia y se ejecute el "else" del operador ternario recibiendo lo que te muestro en la siguiente imagen:

```
→  feriados git:(master) ✗ node index.js
Derrick Reyes no cumple años en ningún día feriado :/
```

Imagen 9. Mensaje por consola indicando que el usuario random no cumple años.
Fuente: Desafío Latam

Ejercicio propuesto (5)

La Pokeapi nos entrega en el siguiente endpoint https://pokeapi.co/api/v2/pokemon/ una propiedad "results" que contiene un arreglo con varios pokemones, en el siguiente código te muestro el arreglo recortado con un solo objeto representando a "bulbasaur".

```
results: [
 {
   name: 'bulbasaur',
   url: 'https://pokeapi.co/api/v2/pokemon/1/',
 }
]
```

Como puedes apreciar, la información de los pokemones no es mucha, solamente el nombre y una url, no obstante dentro de esa url conseguirás una propiedad "weight" que representa el peso y una propiedad "height" qué representa la altura de cada pokemon.

Basado en el ejercicio de los usuarios random y los días feriados, desarrolla una aplicación que devuelva el nombre y el peso de los pokemones. El objetivo será obtener un resultado como el que te muestro a continuación:

```
bulbasaur => Alto: 7 - Peso: 69
ivysaur => Alto: 10 - Peso: 130
venusaur => Alto: 20 - Peso: 1000
charmander => Alto: 6 - Peso: 85
charmeleon => Alto: 11 - Peso: 190
charizard => Alto: 17 - Peso: 905
squirtle => Alto: 5 - Peso: 90
wartortle => Alto: 10 - Peso: 225
blastoise => Alto: 16 - Peso: 855
caterpie => Alto: 3 - Peso: 29
metapod => Alto: 7 - Peso: 99
butterfree => Alto: 11 - Peso: 320
weedle => Alto: 3 - Peso: 32
kakuna => Alto: 6 - Peso: 100
beedrill => Alto: 10 - Peso: 295
pidgey => Alto: 3 - Peso: 18
pidgeotto => Alto: 11 - Peso: 300
pidgeot => Alto: 15 - Peso: 395
rattata => Alto: 3 - Peso: 35
raticate => Alto: 7 - Peso: 185
```

Resumen

A lo largo de esta lectura cubrimos:

- Qué es la asincronía.
- El caso del infierno de los callbacks.
- Ejemplo del uso de un callback, promesas y Async/Await.
- La diferencia de funciones con y sin la palabra reservada Async.
- Ejecutar varias promesas en paralelo.
- Hacer un código más limpio sustituyendo promesas por async await.
- Aplicar Promesas y async/await en un problema planteado.

Solución de los ejercicios propuesto

1. Completa las siguientes frases:

    - Una función que es ejecutada como respuesta de otra se considera un callback

    - El concepto "el infierno de los callbacks" se refiere a una secuencia de varios callbacks anidados

2. Basado en el ejercicio de los sismos, desarrolla una aplicación que utilice una función asíncrona que consulte un usuario random en el siguiente endpoint https://randomuser.me/api/. Al usar el método "then" de esta función llamar a otra función que imprima el nombre y la fecha de nacimiento del usuario recibido.

```
const axios = require('axios')
let nombre
let nacimiento
async function getData() {
 let { data } = await axios.get('https://randomuser.me/api/')
 let user = data.results[0]
 nombre = `${user.name.first} ${user.name.last}`
 nacimiento = user.dob.date
}
function imprimir() {
 console.log(`El usuairo ${nombre} nació el ${nacimiento}`)
}
getData().then(() => {
 imprimir()
})
```

3. Basado en el ejercicio de las promesas que devuelven un número, desarrolla una aplicación con dos promesas que devuelvan tu nombre y tu apellido, y usando el Promise.all imprime la concatenación del resultado de ambas promesas.

```
function promesa1() {
 return new Promise((res, rej) => {
   res('Rafael')
 })
}
function promesa2() {
 return new Promise((res, rej) => {
   res('Aghayev')
 })
}

Promise.all([promesa1(), promesa2()]).then((resultado) => {
 let nombre = resultado[0]
 let apellido = resultado[1]
 console.log(`${nombre} ${apellido}`)
})
```

4. Basado en el ejercicio propuesto anterior, sustituye las promesas por funciones asíncronas y consigue el mismo resultado.

```
async function promesa1() {
 return 'Rafael'
}
async function promesa2() {
 return 'Aghayev'
}

Promise.all([promesa1(), promesa2()]).then((resultado) => {
 let nombre = resultado[0]
 let apellido = resultado[1]
 console.log(`${nombre} ${apellido}`)
})
```

5. La Pokeapi nos entrega en el siguiente endpoint https://pokeapi.co/api/v2/pokemon/ una propiedad "results" que contiene un arreglo con varios pokemones, en el siguiente código te muestro el arreglo recortado con un solo objeto representando a "bulbasaur".

```
const axios = require('axios')

let pokemonesPromesas = []

async function pokemonesGet() {
  const { data } = await axios.get('https://pokeapi.co/api/v2/pokemon')
  return data.results
}

async function getFullData(name) {
  const { data } = await
axios.get(`https://pokeapi.co/api/v2/pokemon/${name}`)
  return data
}

pokemonesGet().then((results) => {
  results.forEach((p) => {
    let pokemonName = p.name
    pokemonesPromesas.push(getFullData(pokemonName))
  })

  Promise.all(pokemonesPromesas).then((data) => {
    data.forEach((p) => {
      console.log(`${p.name} => Alto: ${p.height} - Peso: ${p.weight}`)
    })
  })
})
```