**M2S10: Procesos asíncronos (Promesas y ASYNC/AWAIT)**

**Hola, nos encontramos nuevamente en módulo 2, ¿cómo te sientes?, ¿has aprendido mucho?, espero que tengas muchísima energía y motivación para seguir adelante, no olvides el motivo por el cual estás aquí.**

**¡Sigue adelante, nos vemos! 😄**

**ÍNDICE**

* [Conceptos de asincronismo](https://github.com/U-Camp/BOOT-M2-SEM10/blob/main/README.md#conceptos-de-asincronismo)
* [Callbacks](https://github.com/U-Camp/BOOT-M2-SEM10#callbacks)
* [Promesas](https://github.com/U-Camp/BOOT-M2-SEM10/blob/main/README.md#Promesas)
* [Async / Await](https://github.com/U-Camp/BOOT-M2-SEM10/blob/main/README.md#Async-Await)

**Conceptos de asincronismo**

JavaScript tiene un modelo de procesos:

* Asíncrono.
* No bloqueante. Cuenta con un solo *thread*, no se bloquea. Puedes ejecutar más tareas a pesar de que una en específica tome más tiempo.
* "Event Loop". También conocido como bucle de eventos. Este es el encargado de orquestar toda la arquitectura y de monitorizar si se ha producido algún nuevo evento. De ser así, ejecutará el *callback* correspondiente.

Por ejemplo, casos donde se requiere utilizar un proceso asíncrono es la extracción de datos de un *Backend*. El tiempo que tarda en realizar una conexión con el backend, extraer de base de datos y retornar la petición hará que el resto de las tareas de la aplicación sigan trabajando de manera independiente.

Imagina un proceso asíncrono como una instrucción que se realiza **mientras** se hace otra instrucción. Esto permite manejar múltiples instrucciones a lo largo de un programa.

Para gestionar todas estas instrucciones, existen 3 tipos:

**Callbacks**

Se refiere a una función que se ejecuta dentro de otra función, a partir de la invocación de la primera.

Observa a continuación un ejemplo de callback:

function saludar(nombre) {

console.log("Hola" + " " + nombre )

}

function capturarNombreUsuario (callback) {

let dato = prompt('Ingresa tu nombre')

callback(dato)

}

capturarNombreUsuario(saludar)

* Observa que se ejecutará la función de capturarNombreUsuario.
* Como argumento, se transfiere la función saludar (más no se invoca).
* Obtenemos el dato del usuario y posteriormente pasamos ese dato a la función de saludar a través de la representación callback. Es decir, callback vale saludar y se está usando a través del parámetro.
* Obtenemos como retorno Hola NOMBRE.

Los callbacks nos ayudan a conectar funciones que se invocan en sí mismas para dar mayor control del momento en que necesitan ser llamadas.

**Promesas**

Las promesas son herramientas de los lenguajes de programación que nos sirven para gestionar situaciones futuras en el flujo de ejecución de un programa y nos permiten definir cómo se tratará un dato que sólo estará disponible en un futuro, especificando qué se realizará con ese dato más adelante. Es similar a una promesa en la vida real, una promesa es una garantía de que vamos a hacer algo en el futuro, porque las promesas sólo se pueden hacer en el futuro.

Para esto está la clase estándar Promise. Una **promesa** en JavaScript es una acción asíncrona que puede completarse en algún momento y producir un valor, notificando a cualquiera que esté interesado cuando su valor esté disponible.

La forma más sencilla de crear una promesa es llamando a Promise.resolve. Esta función se asegura de que el valor que le das está envuelto en una promesa. Si ya es una promesa, simplemente la devuelve; de lo contrario, obtiene una nueva promesa que termina inmediatamente con su valor como resultado.

let fifteen = Promise.resolve(15);

quince.then(value => console.log(`Got ${value}`));

// → Got 15

Para obtener el resultado de una promesa, puedes utilizar su método then. Esto registra una función *callback* que será llamada cuando la promesa se resuelva y produzca un valor. Puedes añadir múltiples callbacks a una sola promesa, y serán llamados, incluso si los añades después de que la promesa ya se haya resuelto (terminado).

Pero eso no es todo lo que hace el método then. Devuelve otra promesa, que se resuelve con el valor que devuelve la función returns o, si ésta devuelve una promesa, espera esa promesa y luego se resuelve con su resultado.

Es útil pensar en las promesas como un dispositivo para trasladar valores a una realidad asíncrona. Un valor normal está simplemente ahí. Un valor prometido es un valor que podría estar ya ahí o podría aparecer en algún momento en el futuro. Los cálculos definidos en términos de promesas actúan sobre dichos valores envueltos y se ejecutan de forma asíncrona a medida que los valores están disponibles.

Para crear una promesa, puedes utilizar Promise como constructor. Tiene una interfaz algo extraña: el constructor espera una función como argumento, a la que llama inmediatamente, pasándole una función que puede utilizar para resolver la promesa. Funciona de esta manera, en lugar de, por ejemplo, con un método de resolución, de modo que sólo el código que creó la promesa puede resolverla.

Así es como se crearía una interfaz basada en promesas para la función readStorage

function storage(nest, name) {

return new Promise(resolve => {

nest.readStorage(name, result => resolve(result));

});

}

storage(bigOak, "enemies")

.then(value => console.log("Got", valor));

Esta función asíncrona devuelve un valor significativo. Esta es la principal ventaja de las promesas: simplifican el uso de funciones asíncronas. En lugar de tener que pasar callbacks, las funciones basadas en promesas son similares a las normales: toman la entrada como argumentos y devuelven su salida. La única diferencia es que la salida puede no estar disponible todavía.

Observemos otro ejemplo.

let ejemploPromesa = new Promise((resolve, reject) => {

setTimeout(function(){

resolve("Se ejecutó la promesa")

}, 2000)

ejemploPromesa.then(() => {

console.log("Obtuvimos un mensaje exitoso")

})

})

* Primero, se establece con la palabra new que realizaremos una instancia de Promise, para establecer una promesa.
* Dentro, observa que tendrás una función que como parámetros, tendrá resolve y reject. En la función, deberás establecer si se resuelve o rechaza la función de acuerdo a las operaciones que se vayan decidiendo. En este caso, después de 2 segundos (2000 milisegundos) se ejecutará resolve, dando un resultado exitoso a la promesa.
* Una vez esto, debemos utilizar .then para establecer una consecuencia de un mensaje exitoso, el cual en este caso lo aplicaremos a través de una función como argumento en el cual se ejecute console.log.

Las promesas permiten esperar el resultado de una acción y actuar al respecto.

Es muy común usarse para servicios API porque desconocemos cuánto será el tiempo que tardará en extraer los datos. Por ello, una promesa es una opción óptima.

**ASYNC / AWAIT**

Son esencialmente promesas, pero incluyen "azúcar sintáctico", esto significa que podemoso aplicar una sintaxis más corta y sencilla de entender. No estamos reemplazando, **solo cambiando sintaxis.**

Una función async es una función que sabe cómo esperar la posibilidad de que la palabra clave await sea utilizada para invocar código asíncrono.

Hagamos un ejemplo. Tendremos dos funciones:

* La primera función, es una promesa que devuelve un texto (Datos extraidos) en 2 segundos; la llamaremos realizarLlamadaBD.
* La segunda función, inicia su sintaxis con async. Esto indica que va a resolver procesos asíncronos. Después, en la declaración, observarás que hay una variable resultado, el cual ejecuta la primera función. Como sabemos que va a tardar dos segundos en obtener la respuesta, aplicamos await para esperar ese proceso asíncrono.
* Una vez terminada la espera, se asigna el valor a resultado y seguimos avanzando en la función.
* Al terminar imprimimos el resultado y terminamos.

function realizarLlamadaBD(){

return new Promise (resolve => (

setTimeout(() => {

resolve('Datos extraidos')

}, 2000)

)

)

}

async function consumoDatos(){

console.log("obteniendo...")

const resultado = await realizarLlamadaBD()

console.log("Este es el resultado:", resultado)

return resultado

}

consumoDatos()

En la práctica, utilizarás los tres, dependiendo del contexto y situación en la que te encuentres. Por ello, vale mucho la pena conocerlos a profundidad.

**Hasta aquí el contenido de esta semana, recuerda que puedes consultarlo cada vez que necesites, así como puedes escribirle a tus coaches o participar en las clases de repaso en caso de ser necesario.**

**Nos vemos la siguiente semana. 😊**