



EXERCISES QUE TRABAJAREMOS EN EL CUE:

- EXERCISE 1: FUNCIONES CALLBACK.

EXERCISE 1: FUNCIONES CALLBACK.

¿QUÉ ES EL CÓDIGO ASÍNCRONO?

Según lo revisado anteriormente, éste se refiere al código que se ejecuta y no espera la respuesta para ejecutar el siguiente bloque código. Tiende a ser más efectivo respecto al uso del tiempo, ya que, cuando existe una tarea que demora en responder o terminar, en vez de “bloquearse” esperando la respuesta, puede continuar ejecutando otras tareas hasta que se complete el anterior.

Esta característica puede hacer que los programas sean más rápidos, y que manejen gran cantidad de tareas sin perjudicar la performance del código.

¿QUÉ ES EL LOOP DE EVENTOS DE NODE?

Node utiliza un solo hilo de ejecución para todos sus procesos, y éste es controlado por el loop de eventos, quien se encarga de ir apilando en una lista todas las tareas que deben ser ejecutadas, y “disparar” un evento cada vez que la tarea termina.

El hecho de que Node realice sus tareas en un único hilo, significa que éstas no se ejecutan simultáneamente, sino de manera consecutiva, una detrás de otra. El código asíncrono y el uso del loop de eventos, hacen de Node una herramienta muy potente, capaz de recibir muchas tareas y no bloquear el código cada vez que espera una respuesta.

¿LA EJECUCIÓN DE CÓDIGO ES LINEAL DENTRO DE NODE?

Si bien el código asíncrono tiene la ventaja de aprovechar el tiempo mientras espera una respuesta, este comportamiento puede ocasionar resultados inesperados, que no son tan fáciles de detectar en un principio, y por ello es necesario revisarlo antes de empezar a crear nuestros primeros programas. En la siguiente sección, estudiaremos un par de ejemplos de bloques de código, los cuales nos permitirán entender mejor el comportamiento de Node frente a tareas de tipo asíncronas. Por ahora, solo es necesario tener presente que, si tu código no se ejecuta en el orden que lo esperas, es muy posible que exista alguna tarea asíncrona que aún no tenga su respuesta y, por ende, la ejecución del programa continúe con el siguiente bloque de código.

PREGUNTAS DE CÓDIGO

Tomando en cuenta lo revisado hasta ahora respecto al hilo de ejecución, las funciones de **callback** y peticiones asíncronas, responde las siguientes preguntas observando los bloques de código.

Selecciona el código que implementa una función de **callback**:

1)



```
pregunta1codigo1.js

app.get('/', function(req, res){
  res.send("Status Code 200");
});
```

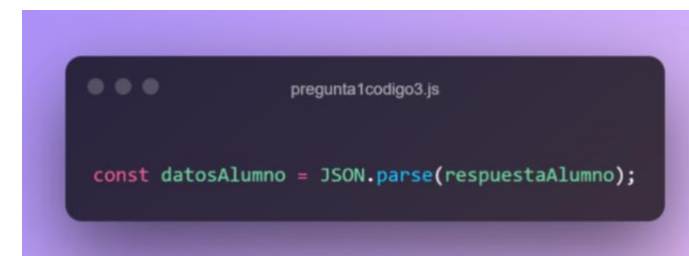
2)



```
pregunta1codigo2.js

function sumaDosNumeros(numero1, numero2) {
  return numero1 + numero2
}
```

3)



```
pregunta1codigo3.js

const datosAlumno = JSON.parse(respuestaAlumno);
```

La respuesta correcta es **la opción 1**. Para entender mejor las funciones de **callback**, podemos ayudarnos de la siguiente manera.

Como ya sabemos, una función de **callback** es aquella que se pasa como argumento de otra. La función que se pasa como argumento, también es conocida como una función anónima, debido a que no lleva nombre en su declaración.

Puede ser que, al principio, la sintaxis para el uso de funciones de **callback** sea un poco confusa. Veamos un ejemplo que ayudará a entender el concepto.



```
ayudaVisual.js

app.get('/', function (req,res){
  res.send('Status Code 200');
});
// Tambien podemos pensar la declaracion de la siguiente forma
// Primero realizamos la definicion de una funcion
function respuesta(req,res){
  res.send('Status Code 200');
}
// Ahora llamamos a la funcion que recibe como uno de sus
// argumentos otra funcion o funcion de callback
app.get('/', respuesta);
```

En la imagen anterior, el primer bloque de código es equivalente al uso del segundo y tercer bloque en conjunto.

El segundo bloque de código declara una función (con el nombre de **respuesta**), y en el tercero se usa una función que recibe como segundo argumento la función **respuesta**, declarada en el bloque anterior. En el primer bloque de código, en vez de declarar una función aparte, solo se realiza la declaración de la función dentro del argumento, empleando una anónima.

¿CUÁL SERÁ EL ORDEN DE APARICIÓN EN CONSOLA DEL SIGUIENTE BLOQUE DE CÓDIGO?

```
pregunta2codigo1.js

const puntajeIdeal = 53;

console.log("obteniendo resultados");

baseDeDatos.findById(idAlumno, function(resultados){
  console.log("los resultados fueron:" + resultados);
});

console.log(puntajeIdeal);
```

1)

```
> obteniendo resultados
> los resultados fueron:
> 53
```

2)

```
> los resultados fueron:
> obteniendo resultados
> 53
```

3)

```
> obteniendo resultados
> 53
> los resultados fueron:
```

Analicemos el código paso a paso.

Primero, se define la variable `puntajeIdeal`, y se le asigna el valor `53`.

Luego, se escribe por consola el mensaje: "obteniendo resultados".

El siguiente paso es hacer una consulta en la base de datos, utilizando un método para obtener los datos de un alumno.

La consulta a la base de datos puede demorar un tiempo determinado, por lo tanto, Node busca la siguiente tarea a ejecutar mientras llega la respuesta.

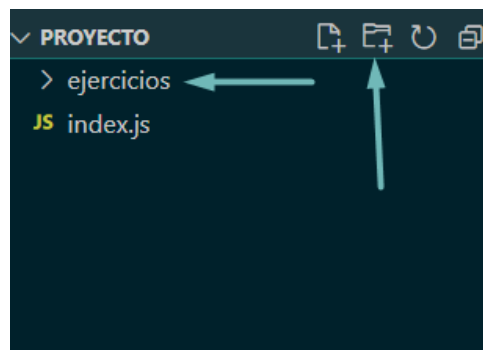
Luego de la consulta, la tarea es escribir por consola el valor de la variable `puntajeIdeal`.

Y ahora que hemos llegado al final de nuestro programa, y ya no quedan más tareas por ejecutar, **Node** simplemente espera la respuesta de la base de datos. Al recibirla, se ejecuta el código contenido en la función de `callback`, es decir, el mensaje "los resultados fueron: 'resultados'".

Una vez analizado nuestro código, podemos observar que **la respuesta correcta es la número 3**.

EJERCICIOS PRÁCTICOS

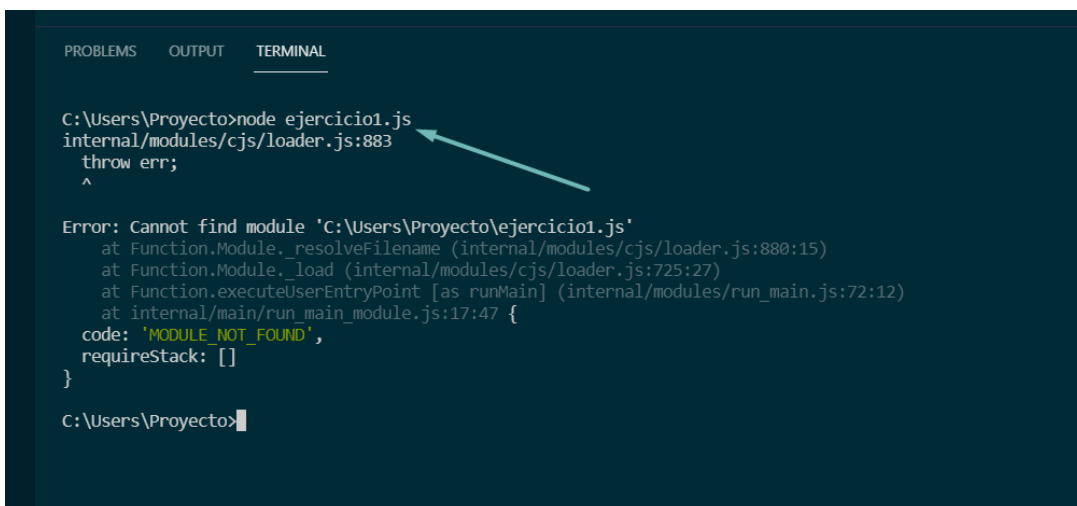
Utilizaremos la misma carpeta generada anteriormente para nuestro primer programa en Node, y en ella crearemos un subdirectorio llamado Ejercicios:



Dentro del subdirectorio, crearemos dos archivos nuevos: `ejercicio1.js`, `ejercicio2.js`. En `ejercicio1.js` escribiremos un simple: `console.log("Soy el ejercicio numero 1")`.



Ejecutaremos `ejercicio1.js` por consola utilizando Node.




Parece que tenemos un error. Resulta que Node primero debe encontrar nuestro archivo, para luego ejecutarlo. Cuando ejecutamos el comando, Node espera que éste se encuentre dentro del directorio en el que se encuentra la terminal, salvo que especifiquemos la ruta del archivo, o nos movamos dentro de la terminal hasta llegar al directorio que contiene el archivo a ejecutar.

Veamos el ejemplo.



Para la primera forma, lo que haremos será pasarle a Node la ruta de nuestro archivo:

```
C:\Users\Proyecto>node ejercicios/ejercicio1.js  
Soy el ejercicio numero 1  
  
C:\Users\Proyecto>
```





O también, movernos en la terminal hacia la carpeta de ejercicios, empleando el comando **cd** (propio de ésta, no tiene que ver con Node), y luego ya podemos ejecutar nuestro archivo utilizando solo su nombre:

```
C:\Users\Proyecto>cd ejercicios  
C:\Users\Proyecto\ejercicios>node ejercicio1.js  
Soy el ejercicio numero 1  
C:\Users\Proyecto\ejercicios>
```



Observa que luego de utilizar el comando **cd**, nuestra ruta se convierte en **Proyecto/ejercicios**. Si quieres volver a correr el archivo **index.js**, debes regresar a la ruta anterior, empleando el comando **"cd .."**.

```
C:\Users\Proyecto\ejercicios>cd ..  
C:\Users\Proyecto>
```



Vamos ahora a nuestro archivo **ejercicio2.js**. Haremos uso de la función **setInterval** de **JavaScript**, para ejecutar una porción de código durante intervalos de tiempo determinado.

Definimos una variable que será nuestro contador para poder mostrar por consola, e inicializaremos su valor en `0`;

```
JS ejercicio2.js X
ejercicios > JS ejercicio2.js > ...
1 // Definicion de variables
2 var contadorDeTiempo = 0;
3
4
5
```


Ahora, creamos la función que mostrará nuestro contador por pantalla.

```
JS ejercicio2.js ●
ejercicios > JS ejercicio2.js > mostrarContador
1 // Definicion de variables
2 var contadorDeTiempo = 0;
3
4 // Definicion de funcion
5 function mostrarContador() {
6     contadorDeTiempo++;
7     console.log("Este es el segundo: " + contadorDeTiempo);
8 }
9
10
```

Ya casi terminamos. Ahora solo nos queda utilizar la función `setInterval`, la cual recibe dos argumentos: una función que ejecuta la porción de código que ya hemos definido; y un número que indica el intervalo de tiempo (en milisegundos) en que esta función será ejecutada.


```
JS ejercicio2.js ●
ejercicios > JS ejercicio2.js > ...
1 // Definicion de variables
2 var contadorDeTiempo = 0;
3
4 // Definicion de funcion
5 function mostrarContador() {
6     contadorDeTiempo++;
7     console.log("Este es el segundo: " + contadorDeTiempo);
8 }
9
10
11 // Declaracion de intervalo de tiempo
12 setInterval(mostrarContador, 1000);
```

Ya podemos ejecutar nuestro código utilizando Node. Recuerda que debes tener en cuenta donde te encuentras posicionado en la terminal, para así saber cómo decirle a Node donde estará tu archivo.



```
PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE
C:\Users\Proyecto\ejercicios>node ejercicio2.js
Este es el segundo: 1
Este es el segundo: 2
Este es el segundo: 3
Este es el segundo: 4
Este es el segundo: 5
Este es el segundo: 6
Este es el segundo: 7
Este es el segundo: 8
Este es el segundo: 9
Este es el segundo: 10
Este es el segundo: 11
Este es el segundo: 12
Este es el segundo: 13
Este es el segundo: 14
Este es el segundo: 15
Este es el segundo: 16
Este es el segundo: 17
Este es el segundo: 18
Este es el segundo: 19
Este es el segundo: 20
```

Ya que no existe ninguna condición que le indique al programa cuando detenerse, nuestro código se ejecutará hasta que demos la orden a Node de que queremos terminar el programa. Para salir de éste, presiona las teclas ctrl + c.