Funciones y Callbacks

Teoría

Introducción

En este apunte aprenderemos algunas particularidades de las funciones en NodeJS, que nos permitirán aprovechar su capacidad al máximo.

Declaración de funciones

Las funciones NodeJS tienen varias particularidades con respecto a otros lenguajes. Veamos todas las formas que ofrece el lenguaje para declarar una función

Estilo clásico

```
function mostrar(params) {
    console.log(params)
}
```

Llamada a la función:

```
mostrar(args)
```

Al ser NodeJS un lenguaje que no requiere especificar el tipo de dato de sus variables (tipado dinámico), tampoco es necesario especificar el tipo de dato que devuelven las funciones, ni el tipo de dato de los parámetros que éstas reciben.

Las funciones *también* son objetos

En NodeJS (y en JavaScript en general), las funciones se comportan como objetos, por lo que es posible asignar una declaración de función a una variable. Si elegimos esta opción, podemos elegir no escribir en forma explícita el nombre en la declaración de la función (esto es, para evitar escribirlo dos veces: en la variable, y en la función).

```
const mostrar = function(params) {
    console.log(params)
}
```

Podemos ejecutarla de la misma manera en que lo hicimos en el ejemplo anterior.

Nuevo estilo (simplificado)

Las últimas versiones del lenguaje nos ofrecen una forma simplificada de declarar una función, cuando la asignamos a una variable, obviando la palabra 'function', y agregando un operador nuevo (esta forma de declarar funciones ofrece además algunas otras características que veremos en mayor profundidad en otros apuntes).

La nueva sintaxis consiste en declarar únicamente los parámetros, y luego conectarlos con el cuerpo de la función mediante el operador => (flecha gorda, o 'fat arrow' en inglés). Veamos un ejemplo:

```
const mostrar = (params) => {
    console.log(params)
}
```

La función se podrá usar de la misma manera que las anteriores.

En el caso de que la función reciba **un solo** parámetro, los paréntesis se vuelven opcionales, pudiendo escribir:

```
const mostrar = params => {
    console.log(params)
}
```

En el caso de que el cuerpo de la función conste de una única instrucción, las llaves se vuelven opcionales, el cuerpo se puede escribir en la misma línea de la declaración (esto es así en muchos otros lenguajes, no solo en NodeJS), y el resultado de computar esa única línea se devuelve como resultado de la función, como si tuviera un "return" adelante. A esto se lo conoce como "return implícito". El ejemplo anterior se vería así:

```
const mostrar = params => console.log(params)
```

Y en este caso la función devolvería "undefined" ya que console.log es de tipo void y por lo tanto no devuelve nada. Un ejemplo, igualmente trivial, pero más explícito, de return implícito sería el siguiente:

```
const promediar = (a, b) => (a + b) / 2
const p = promediar(4, 8) // 6
```

Funciones como parámetros

Como hemos visto, en NodeJS es posible asignar una función a una variable. Esto es porque internamente, las funciones también son objetos (y las variables, referencias a esos objetos).

Es por esto que NodeJS nos permite hacer que una función reciba como parámetro una referencia a otra función.

Veamos un ejemplo, utilizando lo que aprendimos en el punto anterior:

```
const ejecutar = unaFuncion => unaFuncion()
const saludar = () => console.log('saludos')
ejecutar(saludar)
```

Y como ya sabemos: en donde puedo usar una variable, puedo usar también directamente el contenido de esa variable:

```
ejecutar(() => console.log('saludos'))
```

En este ejemplo, la función 'ejecutar' recibe una función ***anónima***, y la ejecuta.
Como es de esperarse, esto también funciona con funciones anónimas con parámetros:

```
const ejecutar = (unaFuncion, params) => unaFuncion(params)
const saludar = nombre => console.log(`saludos, ${nombre}`)
ejecutar(saludar, 'terricola')
```

Callbacks

Un callback es una función que se envía como argumento de otra función, con la intención de que la función que hace de receptora ejecute la función que se le está pasando por parámetro.

De acuerdo a esta definición, podemos decir que la función "*ejecutar*" que usamos en el punto anterior "*recibe un callback*".

Ahora bien, uno de los casos en que más se utiliza este recurso es el siguiente:

Imaginemos que queremos que al finalizar una operación se ejecute un cierto código. Por ejemplo, queremos escribir un archivo, y queremos registrar en un log la hora en se termine de escribir. Es probable que no se pueda saber con exactitud en qué momento va a finalizar. En algunos casos (ya veremos en cuáles) no podemos simplemente ejecutar la operación de

escritura, y luego a continuación, guardar el log. En estos escenarios, las funciones deben recibir como último parámetro un callback, que (por convención) será ejecutado al finalizar la ejecución de la función.

Veamos una función inventada para ver cómo funciona:

```
const formatFecha = f =>
    `${f.getDate()}-${f.getMonth()+1}-${f.getFullYear()}`

// esta es mi función con callback
const escribirArchivo = (ruta, datos, callbackLog) => {
    // acá va la parte en donde se escriben los datos en
    // el archivo, en la ruta especificada.
    // esta operación puede tardar un tiempo indeterminado.
    // al finalizar, ejecuta el código que sigue a continuación

    const fechaString = formatFecha(new Date())
    callbackLog(fechaString, 'grabación exitosa')
}

// esta función será mi callback!
const loguear = (fecha, mensaje) => console.log(`${fecha}: ${mensaje}`)

// así es la llamada a la función
escribirArchivo('/ruta/al/archivo', 'los datos', loguear)
```

Si mi función 'escribirArchivo' fuera la única que utiliza a la otra función 'loguear', en lugar de declararla podríamos usar una *función anónima*, de la siguiente manera:

```
escribirArchivo('/ruta/al/archivo', 'los datos', (fecha, mensaje) =>
    console.log(`${fecha}: ${mensaje}`))
```

Algunas convenciones

Es costumbre dentro de la comunidad de programadores de NodeJS seguir una convención a la hora de realizar funciones con callbacks. En la mayoría de los casos, éstas cumplen con las siguientes características:

- El callback siempre es el **último parámetro**.
- El callback suele ser una función que recibe **dos parámetros**.
- La función llama al callback al terminar de ejecutar todas sus operaciones.

- Si la operación resultó en un **error**, la función llamará al callback pasando el error obtenido como **primer parámetro**.
- Si la operación fue **exitosa**, la función llamará al callback pasando **null** como **primer parámetro**.
- Si la operación (**exitosa**) generó algún **resultado**, éste se pasará al callback como **segundo parámetro**.

Desde el lado del callback, estas funciones deberán saber cómo manejar estos parámetros. Por este motivo, nos encontraremos muy a menudo con la siguiente estructura (ejemplo):

```
const ejemploCallback = (error, resultado) => {
    if (error) {
        // hacer algo con el error!
    } else {
        // hacer algo con el resultado!
    }
}
```

Callbacks anidados

A veces, debemos realizar varias de estas operaciones encadenadas, en serie. Lo que mostramos a continuación, si bien no es muy una buena práctica (y más adelante veremos formas de evitarlo) es un fragmento de código con el que nos podemos encontrar, en el cual una función llama a un callback, y éste a otro callback, y éste a otro, y así sucesivamente. A esto se le conoce como "*callback hell*" (infierno de callbacks).

Ejemplo:

```
const copiarArchivo = (nombreArchivo, callback) => {
    buscarArchivo(nombreArchivo, (error, archivo) => {
        if (error) {
            callback(error)
        } else {
            leerArchivo(nombreArchivo, 'utf-8', (error, texto) => {
                if (error) {
                    callback(error)
                } else {
                    const nombreCopia = nombreArchivo + '.copy'
                    escribirArchivo(nombreCopia, texto, (error) => {
                        if (error) {
                            callback(error)
                        } else {
                            callback(null)
```

```
                        }
                    })
                }
            })
        }
    })
}
```