Universidad Politécnica Salesiana

Nombre: Gabriel Cacuango

Grupo: 1 **Docente:** Ing. Rodrigo Tufiño

Fecha: 2020/04/16

Practica: 02

Fundamentos de NodeJS

En el siguiente trabajo se describen con los aspectos y las definiciones mas importantes de los fundamentos de NodeJS los cuales constaran con los siguientes elementos:

- Let vs Var
- Template literales
- Destructuración
- Funciones de flecha
- Callbacks
- Promesas
- Asyn y Await

Let vs Var

Hay que recalcar la importancia de esta clase ya que ponemos en practica **var** se utiliza para asignar variables de forma global en un script esto quiere decir que se almacena para el uso en cualquier bloque y puede ser invocada y **let** sirve para almacenar variables a un bloque de código respetado los límites de identacion eso quiere decir que si colocamos un let de forma global trabajara para todo el bloque de código que le prosiga.

Codigo	Resultado en pantalla

```
let nombre = "Gabriel ";
if (true) {
    let nombre = "Cacuango";
}
console.log(`Hola ${nombre}`);

// Var permite declarar variables de forma global
for (var i = 0; i <= 5; i++) {
    console.log(`i es igual a : ${i}`);
}

console.log(`el valor final de i ${i}`);
```

```
$ node let-var.js
Hola Gabriel
i es igual a : 0
i es igual a : 1
i es igual a : 2
i es igual a : 3
i es igual a : 4
i es igual a : 5
el valor final de i 6
```

Templates literales

Los templates nos sirven para realizar una combinación especial de incluir texto y variables almacenadas en nuestro código de esta manera no se deben usar con concatenador el signo " + " ya que podemos realizar directamente de la siguiente forma:

`el nombre es: ${getNombre()}`

donde getNombre(), puede ser una variable un parámetro de un objeto o una función como mejor la utilicen.

Codigo	Resultado en pantalla

```
// asisgnacion de variables para bloque
let nombre = "deadpool";
let real = 'Wade Wilson';
console.log(`${nombre} ${real}`);
console.log(nombre + " " + real);

//Asignacion y concatenacion de 2 cadenas
let nombreCompleto = nombre + " " + real;
// Asinacion de templates sirven para unir
// cadenas y concatenarlas con valores de variables
let nombreTemplate = `${nombre} ${real}`;

//comparacion de caracteres
console.log(nombreCompleto === nombreTemplate);

function getNombre() {
    return `${nombre} ${real}`;
}

console.log(`el nombre es : ${getNombre()}`);
```

```
$ node templates.js
deadpool Wade Wilson
deadpool Wade Wilson
true
el nombre es : deadpool Wade Wilson
```

Destructuracion:

Esta opción nos permite renombrar los campos de los atributos de un objeto para luego cambiarlos y trabajar con ellos como deseamos.

Codigo	Resultado en pantalla

```
let deadpool = {
    nombre: 'Wade',
    apellido: 'Wilson',
    poder: 'Regeneracion',
    getNombre: function() {
        return `${this.nombre} ${this.apellido}  - poder: ${thi
    }
}
```

Se define una variable deadpool con los siguientes atributos y un método.

```
Wade Wilson  - poder: Regeneracion
Impresion se la destructuracion
------------------------------
Wade Wilson Regeneracion
```

```
let { nombre: primerNombre, apellido: secondname, poder } =
console.log(primerNombre, secondname, poder);
```

Luego se almacena en otra variable y se cambian los nombres de sus atributos

Función Flecha:

Es una alternativa sistemáticamente mas compacta que permite reducir el tamaño de las funciones y no son adecuadas para ser usadas como constructores además de eso las funciones flechas si tiene una línea de código dentro puede ser reducidas aún más como veremos en el siguiente ejemplo.

Función Flecha	Función Flecha una línea
`let sumar = (a, b) => { return a + b; }`	`let sumar = (a, b) => a + b;`

Ejemplo en clase de la función flecha:

```
let sumar = (a, b) => a + b;
console.log(`la suma de 3 + 4 = ${sumar(3,4)}`);
let saludo = (nombre) => ("Hola " + nombre);
console.log(`Probando mensaje: ${saludo("rodrigo")}`);


let deadpool = {
    nombre: 'Wade',
    apellido: 'Wilson',
    poder: 'Regeneracion',
    getNombre: function() {
            return `${this.nombre} ${this.apellido}  - poder: ${this.poder}`
        }
        // getname:() => `${deadpool.nombre} ${deadpool.apellido} - habilidad: ${deadpool.poder}`
}
console.log(deadpool.getNombre());

// variable nombre = (parametros) =>( lo que quiero)
let promedio = (n1, n2) => ((n1 + n2) / 2);
console.log(`El promedio es: ${promedio(50,100)}`);
```

Obtenemos el siguiente resultado:

```
$ node flecha.js
la suma de 3 + 4 = 7
Probando mensaje: Hola rodrigo
Wade Wilson  - poder: Regeneracion
El promedio es: 75
```

Callbacks

En esta parte de la clase vimos el usa que le damos a un setTimeout con esto logramos realizar una acción luego de cierto tiempo, pero el **callback** lo que realiza es que nos devuelve una respuesta y podemos validarlo que nos devuelva algo cuándo se ejecuta la acción de manera correcta, algo peculiar, al momento de usarlo debemos enviarlo en los argumentos de la función y también es que al momento de enviar al momento de convocar callback (null,…,…) se envía el parámetro null indicando que es un error y que si sale todo bien el error será nulo.

```
let getUsuarioById = (id, nickname, callback) => {
        let n = nickname + " Cacuango"
        let usuario = {
            nombre: n,
            id
        }

        if (id === 20) {
            callback(`El usuario con id ${id} no existe!`);
        } else {
            // se debe mandar un null siemre  enviando primero el error
            callback(null, usuario, 25, 'extra');
        }
    }
```

Luego para realizar el llamado al método se deben tomar ciertas cosas en cuenta, al momento de llamar al callback se debe enviar primero el error indicando en la siguiente imagen y validamos el método para cuando ese error exista caso contrario el programa hará lo que deseamos.

```
    // cuando se utiliza callbacks  de debe enviar err por estandar
getUsuarioById(29, 'Gabriel', (err, usuario, edad, dato) => {
    // si me da un error lo muestro en antalla
    if (err) {
        return console.log(err);
    }
    // si no existe error devuelve  mensaje con un objeto  que es usuario concatenado con la
    console.log("Usuario de la BD:", usuario, `edad: ${edad} parametro extra${dato}`);
});
```

Obtenemos el siguiente resultado en pantalla:

Parámetro Correcto	Parámetro incorrecto

`$ node callbacks.js`
`Usuario de la BD: { nombre: 'Gabriel Cacuango', id: 29 }`
`edad: 25 parametro extraextra`
	`$ node callbacks.js`
	`El usuario con id 20 no existe!`

Otro ejemplo de callback simulamos una base de datos y creamos una función que nos devuelve el elemento empleado usando callback como lo vimos anteriormente:

```
let getEmpleado = (id, callback) => {
    // find nos permite buscar en un objeto como un for
    let empleadoDB = empleados.find(empleado => empleado.id === id);
    if (!empleadoDB) {
        callback(`No existe un empleado con id ${id}`);
    } else {
        callback(null, empleadoDB);
    }
}
```

De igual manera una que obtenga al salió en base al un usuario usando igualmente callback :

```
let getSalario = (empleado, callback) => {
    let salarioDB = salarios.find(salario => salario.id === empleado.id)

    if (!salarioDB) {
        callback(`No se encontró salario para el empleado ${empleado.nombre}`)
    } else {
        callback(null, { nombre: empleado.nombre, salario: salarioDB.salario })
    }
}
```

Y aplicamos los métodos por lo cual debemos validar y tratarlos como callbacks es decir validar para cada uno de ellos el error y en caso que no exista errores nos dará el resultado esperado.

```
getEmpleado(2, (err, empleado) => {
    if (err) {
        return console.log(err);
    }

    getSalario(empleado, (err, respuesta) => {
        // si existe un error lo captura y lo imprime
        if (err) {
            return console.log(err);
        }
        // si no envia una respuesta con el siguiente formato
        console.log(`El salario de ${respuesta.nombre} es de ${respuesta.salario}`);
    })

});
```

Resultados obtenidos:

Parámetro Correcto	Parámetro incorrecto
`$ node callbacks2.js` `El salario de Wilson es de 950`	`$ node callbacks2.js` `No se encontró salario para el empleado Ricardo`

Promesas

Las promesas sustituyen de cierta forma a los callbacks y tienen algunas características las cuales son las siguientes:

- Se debe crear una nueva promesa
- Se deben poner resolve y reject , reject para los errores y resolve cuando todo salga bien.

```
let getEmpleado = (id) => {
        // se crea una nueva promesa  siempre con  resolve y reject
        return new Promise((resolve, reject) => {
            let empleadoDB = empleados.find(empleado => empleado.id === id);
            if (!empleadoDB) {
                // captura el error
                reject(`No existe un empleado con id ${id}`);
            } else {
                // devuelce el resu;tado de estar correcto
                resolve(empleadoDB);
            }
        });
    }
```

De igual forma con la función getSalario lo trabajamos como promesa:

```
let getSalario = (empleado) => {
    return new Promise((resolve, reject) => {
        let salarioDB = salarios.find(salario => salario.id === empleado.id)

        if (!salarioDB) {
            // mensaje de error
            reject(`No se encontró salario para el empleado ${empleado.nombre}`);
        } else {
            // en caso que todo salga bien
            resolve({ nombre: empleado.nombre, salario: salarioDB.salario });
        }

    });
}
```

Luego llamamos a los métodos solo que con la diferencia que luego se deben usar .then para indicarle que esta trabajando con promesas, se almacena en una variable de función flecha y luego usar el catch para que este se encargue de atrapar los errores independientemente si se ejecuta en el getSalario o el getEmpleado de la siguiente forma:

```
getEmpleado(2).then(empleado => {
    return getSalario(empleado);
}).then(resp => {
    console.log(`El salario de ${resp.nombre} es de ${resp.salario}`);
}).catch(err => {
    console.log(err);
});
```

Parámetro Correcto	Parámetro incorrecto

```
$ node promesas.js
El salario de Wilson es de 950
```

```
$ node promesas.js
No se encontró salario para el empleado Ricardo
```

```
$ node promesas.js
No existe un empleado con id 10
```

Async-Away

Gracias a async – away podemos manejar funciones asíncronas de manera más secuencial, brindándole más claridad a nuestro código. Con algunas curiosidades primero siempre debe colocarse la palabra async y el await únicamente dentro de async al momento de ejecutar un método se lo debe tratar como una promesa con el .then y el respectivo catch a continuación vemos un ejemplo tratado en la clase.

```
let getNombre = () => {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve("Gabriel");
        }, 3000);
        // reject("Error al consultar el nombre");
    });
}

let saludo = async() => {
    let nombre = await getNombre();
    return `Hola ${nombre}`;
}

saludo().then(mensaje => {
    console.log(mensaje);
}).catch(err => {
    console.log("Error en el Saludo:", err);
});
```

Lo que realiza es llamar a saludo () lo cual es async guarda en nombre y envía un await a la función getNombre() de esta forma get nombre trabaja con una promesa y envía la palabra Gabriel luego de 3s luego utiliza templates para imprimir un hola con el valor obtenido en getNombre()

Obtenemos los siguientes resultados:

```
$ node async-await.js
Hola Gabriel
```

Conclusiones:

- En los fundamentos de NodeJS se logró revisar varios conceptos importantes y la forma de comprimir nuestro código para que este se encuentre de una manera más elegante y fácil de entender teniendo claro los conceptos tratados en esta práctica.
- La practica se logro implementar ejemplos extras y claros a cerca de los fundamentos de JavaScript ya que esto trabajara de la mano con NodeJS.

Enlace a Github: https://github.com/GabrielCacuango07/Fundamentos-Node