INGENIERIA EN CIENCIAS DE LA COMPUTACIÓN

NOMBRE: OSCAR JIMÉNEZ **CARRERA:** COMPUTACIÓN **ASIGNATURA:** PLATAFORMAS WEB

PRACTICA: 02 **TÍTULO:** FUNDAMENTOS DE NODE.JS

OBJETIVO:

- Entender los fundamentos del lenguaje de programación Node.js

PRE REQUISITOS:

a) Computador con Microsoft Windows o GNU/Linux
b) Interprete de Node.js versión 12 o superior
c) Editor de código fuente Visual Studio Code
d) Repositorio de software Git

INSTRUCCIONES:

1. Lea detenidamente cada uno de los enunciados propuestos
2. Plantee una solución a cada uno de los ejercicios
3. Programe una solución utilizando el lenguaje de programación Node.js
4. Elabore un informe con la solución de los ejercicios

ACTIVIDADES A DESARROLLAR:

1. **Documentar scripts**
   Estudiar, ejecutar y documentar (colocar comentarios) los scripts revisados en clase. Por cada uno de los ejercicios, colocar una descripción en sus propias palabras de los conceptos más importantes. El informe debe contener los siguientes ejercicios:

   - Let vs Var

```
JS let-var.js > ...
1   let nombre = "Wolverine";
2
3   if (true) {
4       let nombre = "Magneto";
5   }
6
7   console.log(`hola ${nombre}`);
8
9   let i;
10  for (i = 0; i <= 5; i++) {
11      console.log(`i = ${i}`);
12  }
13
14  console.log(`valor final de i: ${i}`);
```

Como ya tenemos entendido, con la palabra reservada “var” podemos inicializar una variable de cualquier tipo y de igual forma modificarla después. “Let” tiene la misma funcionalidad la diferencia

es que una variable “let” obligadamente necesita tener un valor definido, en cambio “var” tiene el valor ”undefined” por defecto. Existe otro aspecto muy importante, si tenemos una función y dentro de ella realizamos cualquier bucle y que al finalizar queremos que nos entregue algo, si utilizamos “var” el resultado que nos da nuestro bucle lo podemos utilizar dentro de toda la nuestra función, si a su vez utilizamos “let” este nos restringe y nos permite utilizarlo, en este caso, solo dentro de nuestro bucle, ya que como vimos a un principio una variable “let” debe estar si o si inicializada generando así una restricción mínima al alcance en la variable

Resultado:

```
PS C:\Users\PERSONAL\Desktop\02-fundamentos> node let-var.js
hola Wolverine
i = 0
i = 1
i = 2
i = 3
i = 4
i = 5
valor final de i: 6
```

- **Template literales**

```
JS templates.js > ...
1   let nombre = "Deadpool";
2   let real = 'Wade Wilson';
3
4   console.log(nombre + " " + real);
5   console.log(`${nombre} ${real}`);
6
7   let nombreCompleto = nombre + " " + real;
8   let nombreTemplate = `${nombre} ${real}`;
9
10  console.log(nombreCompleto === nombreTemplate);
11
12  function getNombre() {
13      return `${nombre} ${real}`;
14  }
15
16  console.log(`El nombre es: ${getNombre()}`);
```

El template en NodeJs nos permite llamar alguna variable e imprimirla junto con una frase que nos ayude a saber que es ese dato impreso.
Para ello debemos abrir y cerrar unas comillas especiales y dentro de ellas escribir el mensaje y si queremos añadir una variable la introducimos en llaves anteponiendo el signo de dólar.
Ejemplo:

`El nombre es: $(getNombre)`

Resultado:

```
PS C:\Users\PERSONAL\Desktop\02-fundamentos> node templates.js
Deadpool Wade Wilson
Deadpool Wade Wilson
true
El nombre es: Deadpool Wade Wilson
```

- ## Destructuración

```
JS destructuracion.js > ...
1   let deadpool = {
2       nombre: 'Wade',
3       apellido: 'Wilson',
4       poder: 'Regeneración',
5       getNombre: function() {
6           return `${this.nombre} ${this.apellido} - poder: ${this.poder}`;
7       }
8   }
9
10  // Opcion 1:
11  console.log(deadpool.getNombre());
12
13  let nom = deadpool.nombre;
14  let ape = deadpool.apellido;
15  let pod = deadpool.poder;
16
17  // Opción 2: destructuracion
18  let { nombre: primerNombre, apellido, poder } = deadpool;
19  console.log(primerNombre, apellido, poder);
```

La destructuración es una manera fácil y rápida de llamar a variables que están dentro de una función. Como podemos ver en la imagen en vez de crear nuevas variables y ahí guardar las que están en la función “deadpool”, simplemente la creamos un let donde estarán todas las variables de la función “deadpool”, y si queremos llamarlas simplemente las invocamos con el mismo nombre fueron creadas en la función, o si no podemos asignarles un nuevo nombre en el let creado.

Resultado:

```
PS C:\Users\PERSONAL\Desktop\02-fundamentos> node destructuracion.js
Wade Wilson - poder: Regeneración
Wade Wilson Regeneración
PS C:\Users\PERSONAL\Desktop\02-fundamentos>
```

- ## Funciones de flecha

```
JS flecha.js > ...
1   // Forma habitual
2   function sumar(a, b) {
3       return a + b;
4   }
```

Habitualmente tenemos la costumbre de crear una función con atributos y dentro de esta función proceder hacer la operación que necesitamos.

```
6   // Función Flecha
7   let sumar = (a, b) => a + b;
8
9   console.log(`la suma de 3 + 4 = ${sumar(3,4)}`);
10
11  function saludar() {
12      return "Hola chic@s";
13  }
14
15  let saludo = () => {
16      let a = "Rodrigo";
17      let b = 'Chicos';
18      return `${a} ${b}`;
19  }
20
21  console.log(`${saludo()}`);
22
23  let deadpool = {
24      nombre: 'Wade',
25      apellido: 'Wilson',
26      poder: 'Regeneración',
27      getNombre: function() {
28          return `${this.nombre} ${this.apellido} - poder: ${this.poder}`;
29      }
30  }
31
32  console.log(deadpool.getNombre());
```

La función flecha es una nueva forma de ver las funciones ya que de por si genera un “return” y no hace falta colocarlo. Dentro de una función flecha también podemos generar otra función y para llamar a esta segunda función basta con invocar el nombre de la función seguido de un punto y el nombre de la función a la que estamos haciendo alusión.

Resultado:

```
PS C:\Users\PERSONAL\Desktop\02-fundamentos> node flecha.js
la suma de 3 + 4 = 7
Rodrigo Chicos
Wade Wilson - poder: Regeneración
PS C:\Users\PERSONAL\Desktop\02-fundamentos>
```

- **Callbacks**

```
JS callbacks.js > ...
1   // setTimeout(() => {
2   //     console.log("Hola, muchach@s!");
3   // }, 3000);
4
5   let getUsuarioById = (id, nickname, callback) => {
6       let n = nickname + " perez"
7       let usuario = {
8           nombre: n,
9           id
10      }
11
12      if (id === 20) {
13          callback(`El usuario con id ${id} no existe!`);
14      } else {
15          callback(null, usuario, 25);
16      }
17
18
19  }
20
21  getUsuarioById(10, 'pepito', (err, usuario, edad) => {
22      if (err) {
23          return console.log(err);
24      }
25
26      console.log("Usuario de la BD:", usuario, `edad: ${edad}`);
27  });
```

Se le llama callback en NodeJs, al proceso que se puede generar mientras otro está trabajando. Esto es de mucha ayuda al momento de programa, ya que en ciertos casos existen procesos que necesitan mucho tiempo para ejecutarse, aplicando el concepto callback podemos hacer que mientras se genere un proceso largo, otros más pequeños puedan seguir su curso y ser ejecutados. Aquí podemos ver que la función getUsuario esta buscado el usuario en una base de datos con ID =10 también le está mandando un nombre que es “pepito” y dentro de esta petición de atributos está el callback el cual está generando primero una vista de erros, por si lo llegara a tener, y después está pidiendo el nombre de usuario y su edad. Todo esto lo hace en un solo llamado y una sola función ahorrando varias consultas y varias líneas de código.

Resultado:

```
PS C:\Users\PERSONAL\Desktop\02-fundamentos> node callbacks.js
Usuario de la BD: { nombre: 'pepito perez', id: 10 } edad: 25
PS C:\Users\PERSONAL\Desktop\02-fundamentos>
```

Callbacks:

```
25  let getEmpleado = (id, callback) => {
26      let empleadoDB = empleados.find(empleado => empleado.id === id);
27      if (!empleadoDB) {
28          callback(`No existe un empleado con id ${id}`);
29      } else {
30          callback(null, empleadoDB);
31      }
32  }
33
34  let getSalario = (empleado, callback) => {
35      let salarioDB = salarios.find(salario => salario.id === empleado.id)
36
37      if (!salarioDB) {
38          callback(`No se encontró salario para el empleado ${empleado.nombre}`)
39      } else {
40          callback(null, { nombre: empleado.nombre, salario: salarioDB.salario })
41      }
42  }
43
```

```
45  getEmpleado(3, (err, empleado) => {
46      if (err) {
47          return console.log(err);
48      }
49
50      getSalario(empleado, (err, respuesta) => {
51          if (err) {
52              return console.log(err);
53          }
54
55          console.log(`El salario de ${respuesta.nombre} es de ${respuesta.salario}`);
56      })
57
58  });
59
```

Aquí vamos a emplear los callbacks tanto para la entrega de datos como para las validaciones en el caso de que no se encuentres valores en la base de datos. Como podemos ver en el momento de que no exista un empleado con el ID correspondiente, lo que hacemos es, mediante un"if" preguntar si existe o no ese ID. Si no existe le manda un mensaje de error mediante el callback, si el ID existe le manda el nombre del empleado y un "null", este null es porque el callback recibe dos atributos, el de error y el de respuesta.
En la función salario hacemos lo mismo solo que en este caso vamos a enviar más valores y no solo el Empleado, si no el nombre y el salario. Para ello deberíamos hacer una consulta en la base de datos, en NodeJs simplemente entre llaves se pone el nombre de la fila a que quiero ingresar y dentro de esta el valor que quiero obtener.

Resultado:

```
PS C:\Users\PERSONAL\Desktop\02-fundamentos> node callbacks2.js
No se encontró salario para el empleado Ana
PS C:\Users\PERSONAL\Desktop\02-fundamentos> node callbacks2.js
No existe un empleado con id 30
PS C:\Users\PERSONAL\Desktop\02-fundamentos> node callbacks2.js
El salario de Juan es de 800
PS C:\Users\PERSONAL\Desktop\02-fundamentos>
```

Como observamos, obtenemos distintas respuestas según la consulta, el ejemplo 1 No encontró salario para Ana, en el segundo no ha encontrado en la base el usuario con ID 30 y el tercero si encontró al usuario y el salario por lo que obtenemos su nombre y su salario total.

- **Promesas**

```
let empleados = [{
        id: 1,
        nombre: "Santiago"
    },
    {
        id: 2,
        nombre: "Pepe"
    },
    {
        id: 3,
        nombre: "Ana"
    }
];

let salarios = [{
        id: 1,
        salario: 800
    },
    {
        id: 2,
        salario: 950
    }
];
```

```
25  let getEmpleado = (id) => {
26      return new Promise((resolve, reject) => {
27          let empleadoDB = empleados.find(empleado => empleado.id === id);
28          if (!empleadoDB) {
29              reject(`No existe un empleado con id ${id}`);
30          } else {
31              resolve(empleadoDB);
32          }
33      });
34  }
35
36  let getSalario = (empleado) => {
37      return new Promise((resolve, reject) => {
38          let salarioDB = salarios.find(salario => salario.id === empleado.id)
39
40          if (!salarioDB) {
41              reject(`No se encontró salario para el empleado ${empleado.nombre}`);
42          } else {
43              resolve({ nombre: empleado.nombre, salario: salarioDB.salario });
44          }
45
46      });
47  }
```

Con lo visto anteriormente vemos que para cada consulta debemos tener validadas ciertas peticiones, como mandar un mensaje cuando no se encuentre un empleado o un salario. Para varios programas existen aún más peticiones y se puede utilizar las denominadas “Promesas”.
Lo que tiene en especial estas promesas son dos variables llamadas “resolve” y “reject”. La primea se utiliza cunado la petición a sido aceptada y no ha ocurrido ningún fallo, la segunda por lo contrario devuelve un mensaje de error.

```
64  getEmpleado(20).then(empleado => {
65      return getSalario(empleado);
66  }).then(resp => {
67      console.log(`El salario de ${resp.nombre} es de ${resp.salario}`);
68  }).catch(err => {
69      console.log(err);
70  });
```

Si observamos para que esta función debemos utilizar el método “catch” Este nos permitirá imprimir el error que está ocurriendo gracias a la variable “reject”. De la misma forma para obtener el valor de la función “getSalario” debemos llamarlo con un return y el nombre de la función

Resultado:

```
PS C:\Users\PERSONAL\Desktop\02-fundamentos> node promesas.js
No existe un empleado con id 20
PS C:\Users\PERSONAL\Desktop\02-fundamentos> node promesas.js
No se encontró salario para el empleado Leo
PS C:\Users\PERSONAL\Desktop\02-fundamentos> node promesas.js
El salario de Santiago es de 800
PS C:\Users\PERSONAL\Desktop\02-fundamentos>
```

- **Asyn y Await:**

```
let getNombre = () => {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve("Rodrigo");
        }, 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);
});
```

Todo lo aprendido en este documento es necesario para tener en claro cómo funciona el lenguaje NodeJs, sus funciones y como ejecuta cada una de ella. Aprendido todo eso podemos ver las funciones Async y await, que básicamente nos resumen y hacen más fácil la utilización de los métodos visto anteriormente, En una promesa lo que hacemos es reservar dos variables “resolve” para cuando la petición es correcta y ”reject” cuando ocurre un fallo. Con el método async lo único que hacemos es invocar a “await”, esta última nos devolverá el valor de la función “get Nombre” sin la necesidad que en “getNnombre” exista una validación o no, de si el usuario con ID existe.

Resultado:

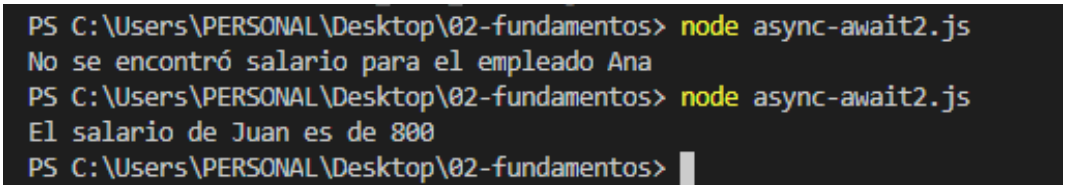
```
PS C:\Users\PERSONAL\Desktop\02-fundamentos> node async-await.js
Hola Rodrigo
```

```
JS async-await2.js > ...
1   let empleados = [{
2           id: 1,
3           nombre: "Juan"
4       },
5       {
6           id: 2,
7           nombre: "Pepe"
8       },
9       {
10          id: 3,
11          nombre: "Ana"
12      }
13  ];
14
15  let salarios = [{
16          id: 1,
17          salario: 800
18      },
19      {
20          id: 2,
21          salario: 950
22      }
23  ];
```

```
47  let getInformacion = async(id) => {
48      let empleado = await getEmpleado(id);
49      let resp = await getSalario(empleado);
50      return `El salario de ${resp.nombre} es de ${resp.salario}`;
51  }
52
53  getInformacion(3)
54      .then(mensaje => console.log(mensaje))
55      .catch(err => console.log(err));
```

En este ejemplo lo tenemos más claro, invocamos la función “async” donde introducimos el ID que queremos buscar, si lo encuentra utilizamos “await” seguido de las funciones “getEmpleado” y “getSalario” ya que ahí se encuentra los valores que buscamos, con un return hacemos la llamada de estos valores y listo.
Al momento de llamar a la función getInformacion con un “then” imprimimos el resultado y si encuentra un error utilizamos un “catch” para imprimirlo. Esto ahorra poner sentencias if’s y que se vuelva un cumulo de código.

Resultado:



2. **GitHub**
   Subir el código revisado y documentado a su cuenta de GitHub. Colocar el enlace del repositorio en el informe de la práctica.

   REPOSITORIO GITHUB:
   https://github.com/ojimenezl/PlataformasWeb7

RESULTADOS OBTENIDOS:

1. El estudiante está familiarizado con la sintaxis y los fundamentos del lenguaje Node.js

CONCLUSIONES:

- El lenguaje de programación NodeJs permite ejecutar operaciones complejas con pocas líneas de código gracias a su estructura y funcionalidades, pero a su vez, esto hace que cierta lógica y costumbres que tenemos al momento de programar cambien de forma drástica. A un principio entender este nuevo paradigma es complicado, pero al momento de llevarlo a cabo y trabajar con él se nota rápidamente una gran diferencia, generando así, una gran utilidad tanto en la programación como en el ahorro de líneas de código.

REFERENCIAS:

[1] F. OpenJS, “Node.js,” nodejs.org. [Online]. Available: https://nodejs.org/en/