

Práctica 02 - Fundamentos de NodeJS

Wilson Aguilar
PLATAFORMAS WEB

17 de abril de 2020

1. Let vs Var

En Javascript tenemos las palabras reservadas `let` y `var` que nos permiten la declaracion de una variable. La diferencia entre estas palabras radica en su alcance.

`let` se limita al scope de donde fue definida, mientras que `var` tiene un alcance global y puede ser llamado fuera del scope.

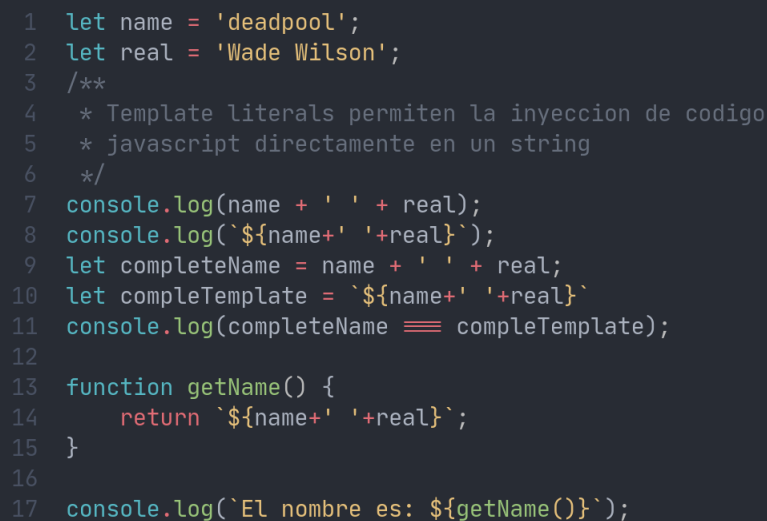


```
1  /**
2   *  let tiene un alcance local
3   */
4  let name = 'Wolverine';
5
6  if (true) {
7      let name = 'Magneto'; // name = magneto
8  }
9
10 console.log(`Hola ${name}`); // name = wolverine
11
12 /**
13  *  var: tiene un alcance global
14  */
15 for (var i = 0; i ≤ 5; i++) {
16     console.log(`i=${i}`);
17 }
18
19 console.log(`valor final de i = ${i}`); // i = 6
```

Figura 1: Ejemplo de let y var

2. Template literals

Los template literals o plantillas de cadena es una nueva forma de definir una cadena, solo que en este caso podemos inyectar directamente una variable o funcion dentro de la cadena de texto.



```
1 let name = 'deadpool';
2 let real = 'Wade Wilson';
3 /**
4  * Template literals permiten la inyeccion de codigo
5  * javascript directamente en un string
6  */
7 console.log(name + ' ' + real);
8 console.log(`${name} ${real}`);
9 let completeName = name + ' ' + real;
10 let completeTemplate = `${name} ${real}`
11 console.log(completeName === completeTemplate);
12
13 function getName() {
14     return `${name} ${real}`;
15 }
16
17 console.log(`El nombre es: ${getName()}`);
```

Figura 2: Ejemplo de template literals

Los template literals son muy usados en frameworks de Javascript enfocados al front-end. Lo utilizan para crear un fragmento o template de un componente de HTML y luego inyectan la variable que va a ser dinámica directamente en el string, lo que hace que el código sea mejor legible.

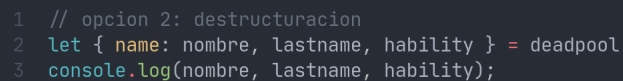
3. Destructuración

La destrucutracion nos permite extraer variables que se encuentran dentro de un objeto.



```
1 let deadpool = {
2   name: 'Wade',
3   lastname: 'Wilson',
4   hability: 'regeneration',
5   getName: function () {
6     return `${this.name} ${this.lastname}
7   } - hability: ${this.hability}`;
8 }
9 /**
10  * La destructuracion nos permite la extraccion de variables o funciones de un objeto.
11  */
12 // Si queremos obtener las propiedades de un
13 // objeto por separado
14
15 // opcion1: de manera normal
16 console.log(deadpool.getName());
17 let n = deadpool.name
18 let ln = deadpool.lastname
19 let pow = deadpool.hability
```

Figura 3: Extracción de variables sin destructuración



```
1 // opcion 2: destructuracion
2 let { name: nombre, lastname, hability } = deadpool
3 console.log(nombre, lastname, hability);
```

Figura 4: Destructuración

4. Funciones de flecha

Las funciones de flecha es una nueva forma de escribir funciones, con este método podemos reducir un poco las lineas de código y es un poco mas legible.

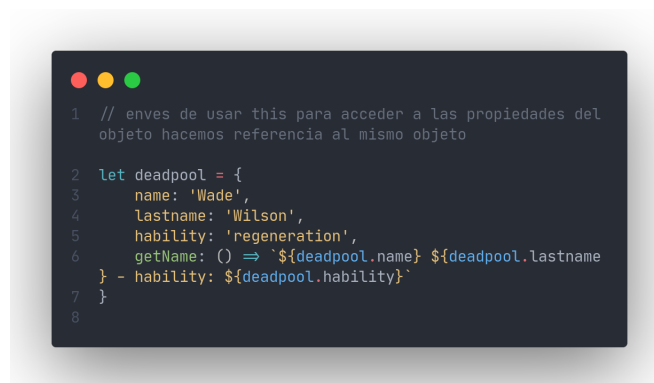


```
1 // Funcion de manera habitual
2 function saludar() {
3   return 'Hola chicos';
4 }
5
6 // Funcion de flecha
7 let saludo = () => 'Hola chicos';
```

Figura 5: Funciones de flecha.

4.1. Problemas

Al usar `this` con las funciones de flecha dentro de un objeto literal, hacemos referencia al objeto que engloba a todo el sistema y no al objeto literal en si. En ese caso es mejor usar las funciones de la manera habitual con la palabra reservada `function`.

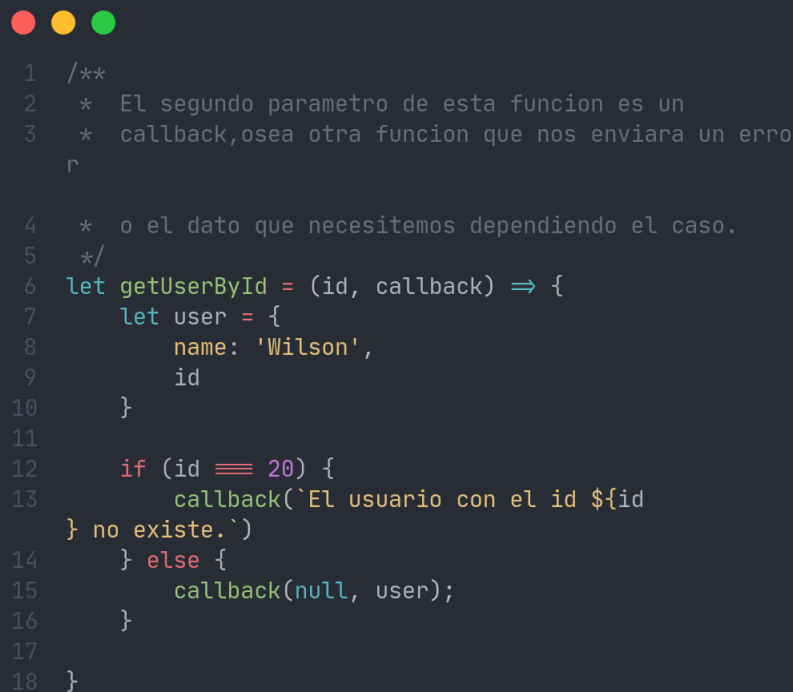


```
1 // enves de usar this para acceder a las propiedades del
  // objeto hacemos referencia al mismo objeto
2
3 let deadpool = {
4   name: 'Wade',
5   lastname: 'Wilson',
6   hability: 'regeneration',
7   getName: () => `${deadpool.name} ${deadpool.lastname}
8   } - hability: ${deadpool.hability}`
9 }
```

Figura 6: Referencia al objeto envés de usar `this`.

5. Callbacks

La asincronía de javascript trae muchos beneficios, pero también un par de problemas y es que los procesos que queremos ejecutar de manera sincrónica es un poco difícil. Para solucionar esto tenemos los callbacks que simplemente es ejecutar una función dentro de otra.



```
1  /**
2   * El segundo parametro de esta funcion es un
3   * callback, osea otra funcion que nos enviara un erro
4   * o el dato que necesitamos dependiendo el caso.
5   */
6  let getUserById = (id, callback) => {
7    let user = {
8      name: 'Wilson',
9      id
10    }
11
12    if (id === 20) {
13      callback(`El usuario con el id ${id}
14    } no existe.`)
15    } else {
16      callback(null, user);
17    }
18  }
```

Figura 7: Definición de un callback.

Para usar el callback lo usamos igual que una función, solo que en el argumento de callback en vez de enviar una variable enviamos una función que se encargara de manejar los datos enviados, en este caso el usuario.

```

1  /**
2   * Al usar la funcion el segundo parametro nos va a
3   * devolver un error o los datos del usuario.
4   */
5  getUserById(20, (err, user) => {
6      if (err) {
7          return console.log(err);
8      }
9      console.log('Usuario de la base de datos es: ',
10         user);
11 });

```

Figura 8: Uso del callback

5.1. Problemas

Los callbacks tienen un problema y es que cuando queremos ejecutar varias acciones de manera sincrónica, el código empieza a verse poco legible y se empieza a formar una especie de cascada.

```

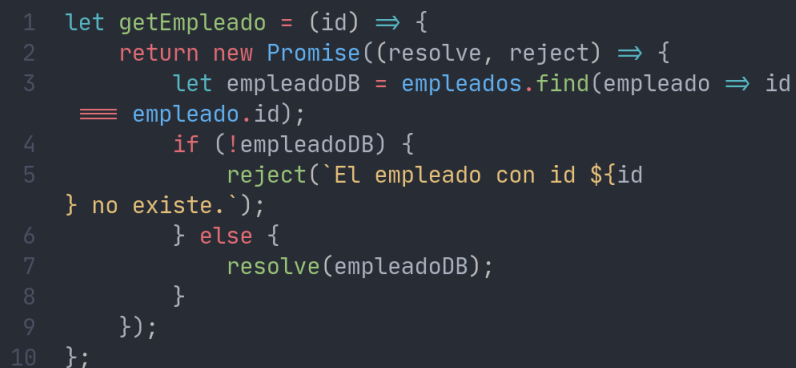
1  /**
2   * Encadenando de callbacks
3   */
4  getEmpleado(2, (err, empleado) => {
5      if (err) {
6          return console.log(err);
7      }
8
9      getSalario(empleado, (err, salario) => {
10         if (err) {
11             return console.log(err);
12         }
13
14         console.log(salario);
15     });
16 });

```

Figura 9: Callbacks en cadena

6. Promesas

Las promesas en javascript son una alternativa a los callbacks. Nos permiten hacer procesos asíncronos de manera síncrona. El cambio radica es que ya no vamos a recibir un callback en la función, solo recibirá el parámetro normal. La función devolverá una nueva promesa y para devolver el resultado de la promesa usamos `resolve()`, en caso de que queramos enviar un error usamos `reject()`.



```
1 let getEmpleado = (id) => {
2   return new Promise((resolve, reject) => {
3     let empleadoDB = empleados.find(empleado => id
4     === empleado.id);
5     if (!empleadoDB) {
6       reject(`El empleado con id ${id
7       } no existe.`);
8     } else {
9       resolve(empleadoDB);
10    }
11  });
12 }
```

Figura 10: Ejemplo de promesas

Para ejecutar promesas tenemos el método `.then()` que recibe 2 callbacks, el primero se encarga de manejar la promesa resuelta o el `resolve` y el segundo se encarga de manejar cuando se ejecuta el `reject` de la promesa.



```

1  /* Promesas de manera normal */
2
3  getEmpleado(4).then(
4    empleado => {
5      getSalario(empleado).then(
6        salario => {
7          console.log(salario);
8        },
9        err => {
10         console.log(err);
11       }
12     )
13   },
14   err => {
15     console.log(err);
16   }
17 );

```

Figura 11: Ejecucion de varias promesas.

También tenemos otra forma de manejar las promesas, y es que las podemos encadenar lo que nos permite ahorrar algunas líneas de código. Enés de ejecutar una promesa dentro de otra lo que hacemos es devolver la ejecución de otra promesa, así se van encadenando y para manejar errores que se produzcan en cualquiera de las promesas usamos el método `.catch()` una sola vez.



```

1  /* Promesas en cadena */
2
3  getEmpleado(1).then(
4    empleado => {
5      return getSalario(empleado).then(
6        salario => {
7          console.log(salario);
8        }
9      );
10   }
11 ).catch(err => {
12   console.log(err);
13 });

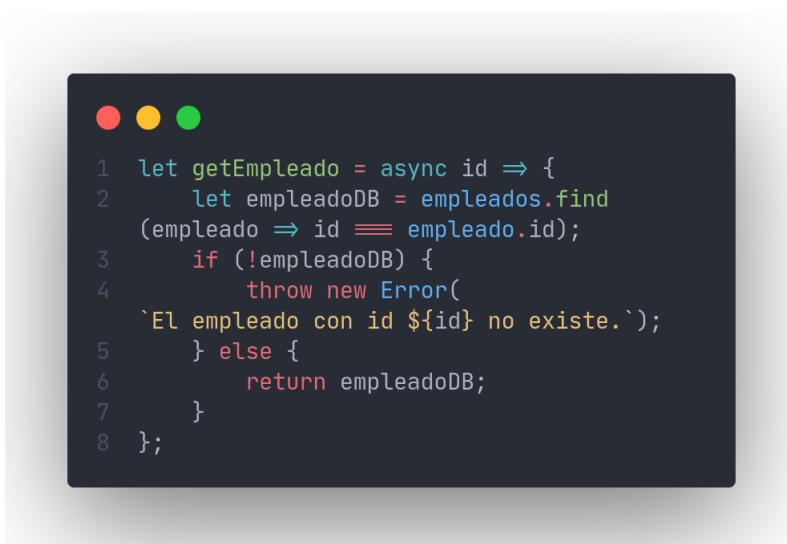
```

Figura 12: Encadenamiento de promesas.

7. Async - Await

Las funciones `async` y `await` nos permiten manejar las promesas de una manera mucho mejor.


`Async` nos permite definir una función asíncrona que como resultado nos va a devolver una promesa que podrá ser manejada como tal. El `resolve` de esta función será igual a hacer un simple `return` y en caso de querer hacer un `reject` lanzamos un error con `throw new Error()`.

A screenshot of a code editor with a dark background and light-colored text. The code defines an asynchronous function named `getEmpleado` that takes an `id` parameter. Inside the function, it uses `empleados.find` to search for an employee by ID. If the employee is not found, it throws an error with a message. Otherwise, it returns the employee object. The code is numbered from 1 to 8.

```
1 let getEmpleado = async id => {
2   let empleadoDB = empleados.find
3   (empleado => id === empleado.id);
4   if (!empleadoDB) {
5     throw new Error(
6       `El empleado con id ${id} no existe.`);
7   } else {
8     return empleadoDB;
9   }
10 }
```

Figura 13: Ejemplo de async functions


Para manejar varias funciones `async` que simulen un proceso síncrono podemos usar el `await` que espera a que la promesa sea resuelta y nos devuelve el valor de esa promesa que la podemos guardar en una variable. Este operador `await` solo puede ser usado dentro de una función `async`.



```
1 const getInformacion = async (id) => {  
2   let empleado = await getEmpleado(id);  
3   let resp = await getSalario(empleado);  
4   return `El salarop de ${resp.nombre} es de ${resp.salario}`;  
5 }
```

Figura 14: Ejemplo de await

En caso de que suceda un error podemos manejarlo de manera que fuera una promesa con el metodo `.catch()`.



```
1 getInformacion(4)  
2   .then(msg => console.log(msg))  
3   .catch(err => console.log(err));
```

Figura 15: Capturando error en async function