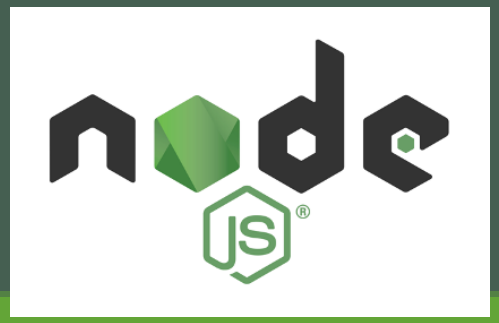


Módulos

Contenido

- callBacks
- EventEmitter

CallBacks: Conceptos

- En JavaScript se usa la **notación CPS** (abreviatura de continuation-passing style) para escribir el código de un programa en el que las Continuaciones se escriben y se pasan de forma explícita.
- Cuando se escribe un programa en notación CPS, cada función recibe un parámetro adicional, que representa la Continuación de la función. En lugar de retornar, la función invocará la continuación recibida pasando el valor de retorno. De esta forma, las funciones nunca regresan al código que las llamó, sino que la ejecución del programa transcurre “hacia adelante” sin retornar hasta que el programa finalice.
- CPS es un concepto general y no siempre está asociado a programación asíncrona.

CallBacks: Conceptos

- Las funciones callbacks, por tanto, sustituyen a la instrucción **return** que requiere la ejecución síncrona.
- En Node todas las APIs soportan el uso de callbacks

```
//CallBacks en programación síncrona

function add(a, b, callback) {
callback(a + b);
}

console.log('before');
add(1, 2, function(result) {
console.log('Result: ' + result);
});
console.log('after');
//Resultado

before
Result: 3
after
```

```
//CallBacks en programación asíncrona

function addAsync(a, b, callback) {
setTimeout(function() {
callback(a + b);
}, 100);
}

console.log('before');
addAsync(1, 2, function(result) {
console.log('Result: ' + result);
});
console.log('after');
//Resultado
before
after
Result: 3
```

CallBacks: Conceptos

- Ej: Lectura de un fichero en **modo síncrono**:
  - Crear un fichero input.txt con un texto de prueba, en el directorio de un proyecto: ejemploCallBacks
  - Crear un fichero main.js que importe el módulo fs
  - Volcar el contenido del fichero a una variable con el método:

```
var data = fs.readFileSync('input.txt');
```

  - Imprimir el contenido del fichero por consola.
  - Imprimir un mensaje por consola ('Fin del programa').

CallBacks: Conceptos

- Ej: Lectura de un fichero en **modo asíncrono**:
  - Crear un fichero input.txt con un texto de prueba, en el directorio de un proyecto: ejemploCallBacks
  - Crear un fichero main.js que importe el módulo fs
  - Leer el fichero con el método:

```
 fs.readFile('input.txt', function (err, data) {
   if (err) return console.error(err);
   console.log(data.toString());
});
```

  - Imprimir un mensaje por consola ('Fin del programa').

Convenciones de Callbacks en NodeJS

- Las funciones callback siempre son el último argumento de la función. Ej:

```
fs.readFile(filename, [options], callback)
```

- Los errores siempre se sitúan al principio de la función callback y deben ser del tipo error. Ej:

```
fs.readFile('foo.txt', 'utf8', function(err, data) {
if(err)
handleError(err);
else
processData(data);
});
```

- En CPS asíncrono los errores se tienen que propagar a la siguiente función de callback. Ej:

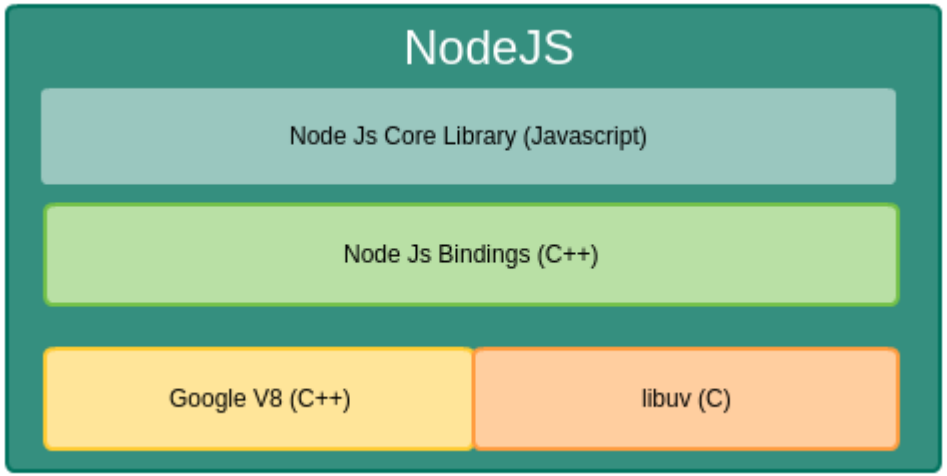
```
var fs = require('fs');
function readJSON(filename, callback) {
fs.readFile(filename, 'utf8', function(err, data) {
var parsed;
if(err)
//propagate the error and exit the current function
return callback(err);
try {
//parse the file contents
parsed = JSON.parse(data);
} catch(err) {
//catch parsing errors
return callback(err);
}
//no errors, propagate just the data
callback(null, parsed);
});
};
```

Ejercicio 5

- Crear una función que pase el contenido de un array a un callback y éste genere un nuevo array que multiplique cada valor del array anterior por 2.
  - TIPS: imitar a la función array.map

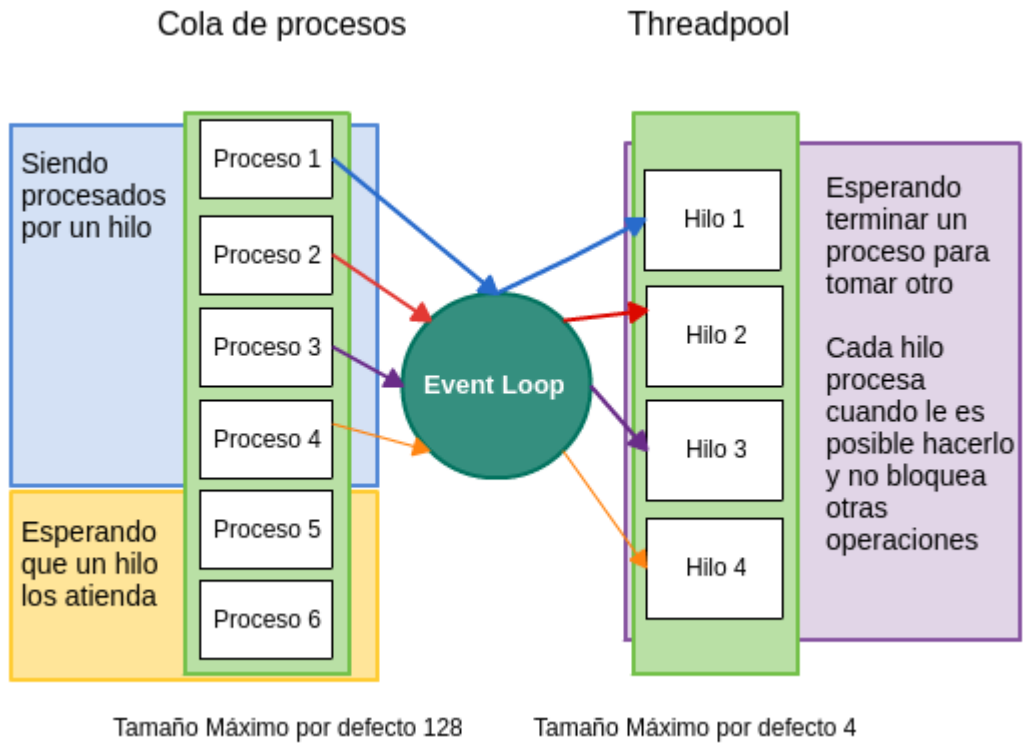
Event Loop

- Es un único subproceso que realiza todas las operaciones de entrada y salida (I/O) de forma asíncrona.
- Es una cola de funciones. Cuando se ejecuta una función asíncrona, la función devuelve el código interno de la función, lo envuelve y se inserta en una cola.
- El motor de JavaScript manda las operaciones a la cola y hace que se procesen en segundo plano para no bloquear las demás operaciones
- Node.js usa una librería llamada **libuv**, la cual proporciona una manera de añadir las operaciones necesarias a la cola de forma asíncrona (conocida como **threadpool**) y estas operaciones corren de forma nativa en el SO.



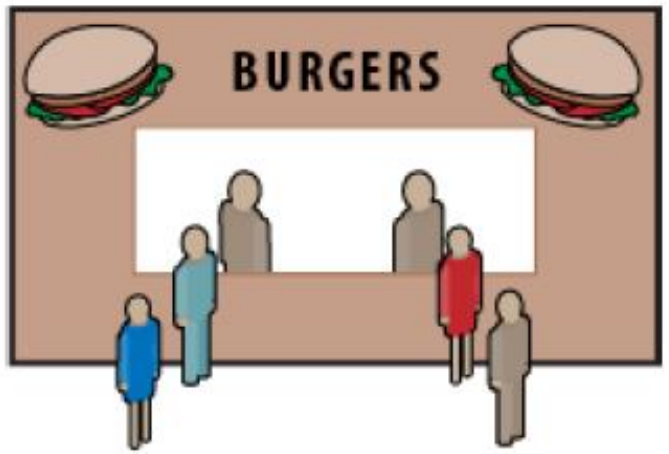
Event Loop: Proceso

- El funcionamiento se da de la siguiente manera:
  1. Existe una cola de tareas
  2. Se define el número de hilos
  3. Cada hilo toma una tarea y la ejecuta
  4. Una vez que la tarea esté completa, se disparará otra
  5. Si no hay tareas, el subproceso permanece inactivo
  6. Si encuentra una nueva tarea en cola se inicia el procesamiento



Programación Orientada a Eventos (Event-Driven Programming)

- En NodeJS, tan pronto como se inicia el servidor se inician variables, se declaran funciones y se espera a que ocurra algún evento
- En una aplicación orientada a eventos, el loop principal responde a los eventos lanzando la función de callback cuando se detecta un evento
- En una aplicación de servidor existen multitud de eventos: peticiones HTTP, consultas de bd, acceso ficheros,...
- Ejemplo Restaurante:



EDP

- Ej.: Programación de un evento de bloqueo I/O. Señalar cuando termina una consulta a una bd:

```
result = query('SELECT * FROM posts WHERE
id = 1');
do_something_with(result);
```

- En EDP:

```
query_finished = function(result) {
do_something_with(result);
}
query('SELECT * FROM posts WHERE id = 1',
query_finished);
```

Patrón Observer

- Define un objecto llamado **subject** que puede notificar a un grupo de **observers** (o **listeners**) cuando ocurre un cambio de estado
- La diferencia principal con el patron de callback es que el sujeto puede notificar a muchos observadores mientras que un callback es el único listener de la finalización de una tarea asíncrona.
- Las funciones que escuchan los eventos actúan como observadores. Cada vez que un evento se dispara, su función de escucha comienza a ejecutarse.
- Node.js tiene varios eventos incorporados disponibles a través del módulo de eventos y la clase EventEmitter que se utilizan para vincular eventos y events-listeners.

Event Emitter

- Crear un objeto eventEmitter:

```
//Módulo de eventos de importación
var events = require('events');

//Crear un objeto eventEmitter
var eventEmitter = new events.EventEmitter();
```

- Enlazar un controlador de eventos con un evento:

```
//Vincular evento y controlador de eventos
eventEmitter.on('eventName', eventHandler);
```

- Lanzar un evento programáticamente de la siguiente manera:

```
eventEmitter.emit('eventName');
```

Event Emitter: Ejemplo

- Crear un archivo js llamado main.js con el siguiente código:

```
// Import events module
var events = require('events');
// Create an eventEmitter object
var eventEmitter = new events.EventEmitter();
// Create an event handler as follows
var connectHandler = function connected() {
    console.log('connection succesful.');
// Fire the data_received event
eventEmitter.emit('data_received'); }
// Bind the connection event with the handler
eventEmitter.on('connection', connectHandler);
// Bind the data_received event with the anonymous function
eventEmitter.on('data_received', function(){
    console.log('data received succesfully.'); });
// Fire the connection event
eventEmitter.emit('connection');
console.log("Program Ended.");
```

Event Emitter: Ejemplo

- Al ejecutar debe producir el siguiente resultado:

```
connection successful.
data received successfully.
Program Ended.
```

Ejercicio 6

- Crear un objeto EventEmitter que emita la fecha actual (Date.now()) dentro de una función setInterval() cada medio segundo.
- Asociar un listener a este evento que imprima la fecha por pantalla.

Ejercicio 7

- Recrear el ejercicio anterior con callback

Event Emitter: Reto

- Crear una función que acepte un callback y devuelva un EventEmitter