



**Clase 28.** Programación Backend

# ***Global & Child process***



## ***OBJETIVOS DE LA CLASE***

- Conocer acerca del objeto *process*.
- Aplicar *child\_process* y sus métodos.
- Comprender los procesos en NodeJS.

# ***CRONOGRAMA DEL CURSO***

Clase 27



**Proceso principal del  
servidor**

Clase 28



**Global & Child Process**

Clase 29



**Clusters y Escalabilidad**

# ***GLOBAL PROCESS***

# ***OBJETO PROCESS***



# ***Objeto Process***

- Como ya hemos visto, el objeto process es una variable global disponible en NodeJS que nos ofrece diversas informaciones y utilidades acerca del proceso que está ejecutando un script Node.
- Contiene diversos métodos, eventos y propiedades que nos sirven no solo para obtener datos del proceso actual, sino también para controlarlo.
- Al ser un objeto global quiere decir que lo puedes usar en cualquier localización de tu código NodeJS, sin tener que hacer el correspondiente require().



# ***Datos del proceso***

Algunos ejemplos de los datos del proceso que se pueden consultar con el objeto process.

```
'Directorio actual de trabajo:' + process.cwd();  
'Id del proceso:' + process.pid;  
'Versión de Node:' + process.version;  
'Título del proceso:' + process.title;  
'Sistema operativo:' + process.platform;  
'Uso de la memoria:' + process.memoryUsage();
```



# ***Salir de la ejecución***

- A veces, se necesita salir de la ejecución de un programa en Node. Esto lo podemos conseguir mediante el método **exit** del objeto *process*.

```
process.exit();
```

- Provocará que el programa acabe, incluso en el caso que haya operaciones asíncronas que no se hayan completado o que se esté escuchando eventos diversos en el programa.

👉 El método exit puede recibir opcionalmente un código de salida. Si no indicamos nada se entiende "0" como código de salida.

```
process.exit(3);
```





# Función ***.on()***

- La mayor funcionalidad de *process* está contenida en la función ***.on()***.
- Dicha función está escuchando durante todo el proceso que se ejecuta, es por eso que solo se puede actuar sobre su callback.
- Se define como se definen los eventos en Javascript. En el método *on*, indicando el tipo de evento que queremos escuchar y un *callback* que se ejecutará cuando ese evento se dispare.

```
process.on("evento",callback);
```



# ***Evento 'beforeExit'***

- Normalmente, el proceso de Node se cerrará cuando no haya trabajo programado, pero un oyente registrado en el evento *beforeExit* puede realizar llamadas asincrónicas y, por lo tanto, hacer que el proceso de Node continúe.
- No debe usarse como una alternativa al evento de *exit* a menos que la intención sea programar trabajo adicional.

```
process.on('beforeExit', (code) => {  
  console.log('Process beforeExit event with code: ', code);  
});
```

# Evento 'exit'



- El evento *exit* se emite cuando el proceso de Node está a punto de salir como resultado de que:
  - El método *process.exit()* se llama explícitamente.
  - El ciclo de eventos de Node ya no tiene ningún trabajo adicional que realizar.
- No hay forma de evitar la salida del bucle de eventos en este punto, y una vez que todos los oyentes de 'salida' hayan terminado de ejecutar, el proceso de Node terminará.

```
process.on('exit', (code) => {  
  console.log(`About to exit with code: ${code}`);  
});
```

# ***Evento 'uncaughtException'***



- Se emite cuando una excepción es devuelta hacia el bucle de evento.
- Si se agregó un listener a esta excepción, no se producirá la acción por defecto (imprimir una traza del stack y salir).
- Es un mecanismo muy básico para manejar excepciones.

```
process.on('uncaughtException', function (err) {  
  console.log('Excepción recogida: ' + err);  
});  
  
setTimeout(function () {  
  console.log('Esto seguirá ejecutándose.');}, 500);  
  
// Se fuerza una excepción, pero no se recoge.  
nonexistentFunc();  
console.log('Esto no se ejecutará.');
```

# ***'process.execPath'***



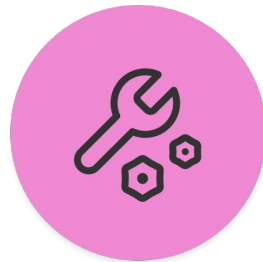
- Esta propiedad devuelve el nombre de la ruta absoluta del ejecutable que inició el proceso Node. Los enlaces simbólicos, si los hay, se resuelven.
- Ejemplo de ruta: `'/usr/local/bin/node'`

# ***'process.stdout.write'***



- La propiedad *process.stdout* devuelve una secuencia conectada a stdout.
- Es un stream de escritura para stdout.
- Ejemplo de la definición de *console.log*:

```
console.log = function (d) {  
    process.stdout.write(d + '\n');  
};
```



# ***USO DEL OBJETO PROCESS***

*Tiempo: 10 minutos*



# ***Uso del objeto Process***

*Tiempo: 10 minutos*

Realizar una aplicación en Node.js que permita recibir como parámetros una cantidad ilimitada de números, con los cuales debe confeccionar el siguiente objeto (se imprimirá por consola):

```
{
  datos: {
    numeros: [los, números, ingresados],
    promedio: (el promedio entre ellos),
    max: (valor máximo),
    min: (valor mínimo),
    ejecutable: (nombre del ejecutable),
    pid: (process id)
  }
}
```





# ***Uso del objeto Process***

*Tiempo: 10 minutos*

En el caso de ingresar un número no válido, se creará un objeto de error con el siguiente formato (se imprimirá por consola):

```
{
  error: {
    descripcion: 'error de tipo',
    numeros: [array, de, entrada],           // ej. [1,2,'pepe',4, true]
    tipos: [array, con, tipos, de, entrada] // ej. [number,number,string,number,boolean]
  }
}
```

En este caso de error, la aplicación saldrá con código de error -5



# ***Uso del objeto Process***

*Tiempo: 10 minutos*

Si no ingresó ningún número, el objeto de error será:

```
{  
  error: {  
    descripcion: 'entrada vacía'  
  }  
}
```

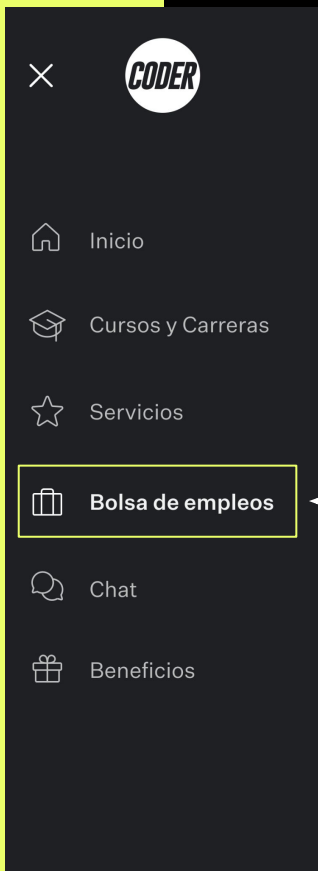
En este caso de error, la aplicación saldrá con código de error **-4**

En los casos de error, se representará en consola el código antes de finalizar.



***BREAK***

**¡5/10 MINUTOS Y VOLVEMOS!**



Nuevo

# ¡Lanzamos la Bolsa de Empleos!

Un espacio para seguir **potenciando tu carrera** y que tengas más **oportunidades de inserción laboral**.

Podrás encontrar la **Bolsa de Empleos** en el menú izquierdo de la plataforma.

Te invitamos a conocerla y ¡postularte a tu futuro trabajo!

Conócela

# ***CHILD PROCESS***

# ***Single Thread (único hilo)***



- Cuando ponemos en marcha un programa escrito en NodeJS se dispone de un único hilo de ejecución.
- Una ventaja de esto es que permite atender mayor demanda con menos recursos.
- Todas las operaciones que NodeJS no puede realizar al instante (operaciones no bloqueantes), liberan el proceso, es decir, se libera para atender otras solicitudes.
- El hilo principal podrá estar atento a solicitudes, pero una vez que se atiendan, Node podrá levantar de manera interna otros procesos para realizar todo tipo de acciones que se deban producir como respuesta a esas solicitudes. Estos procesos secundarios pueden crearse con el módulo *child\_process*.

# Proceso hijo



- Un proceso hijo es un proceso creado por un proceso padre.
- Node nos permite ejecutar un comando del sistema dentro de un proceso hijo y escuchar su entrada / salida.
- Los desarrolladores crean de forma habitual procesos secundarios para ejecutar comandos sobre su sistema operativo cuando necesitan manipular el resultado de sus programas Node con un shell.
- Podemos crear procesos hijo de 4 formas diferentes:

`exec()`

`spawn()`

`execFile()`

`fork()`

# ***FORMAS DE CREAR PROCESOS HIJO***



# Proceso secundario con **'exec( )'**



- Requerimos el comando `exec` del módulo `child_process`.
- En la ejecución de la función `exec`, el primer argumento es el comando `ls-lh`. Este, enumera todos los archivos y carpetas del directorio actual en formato largo, con un tamaño total de archivo en unidades legibles por el ser humano en la parte superior del resultado.

```
const { exec } = require('child_process');  
  
exec('ls -lh', (error, stdout, stderr) => {  
  if (error) {  
    console.error(`error: ${error.message}`);  
    return;  
  }  
  
  if (stderr) {  
    console.error(`stderr: ${stderr}`);  
    return;  
  }  
  
  console.log(`stdout: \n${stdout}`);  
});
```

# Proceso secundario con **'exec( )'**



- El segundo argumento es el *callback*, el cual a su vez tiene 3 parámetros.
- Si el comando no se ejecuta, se imprime el motivo en **error**.
- Si el comando se ejecutó correctamente, cualquier dato que escriba al flujo de resultado estándar se captura en **stdout** y cualquier dato que escriba al flujo error estándar se captura en **stderr**.

```
const { exec } = require('child_process');  
exec('ls -lh', (error, stdout, stderr) => {  
  if (error) {  
    console.error(`error: ${error.message}`);  
    return;  
  }  
  
  if (stderr) {  
    console.error(`stderr: ${stderr}`);  
    return;  
  }  
  
  console.log(`stdout: \n${stdout}`);  
});
```

# Proceso secundario con **'exec( )'**



- Al ejecutar el archivo en la terminal el output será como el que se muestra en la imagen.

```
Output
stdout:
total 4.0K
-rw-rw-r-- 1 sammy sammy 280 Jul 27 16:35 listFiles.js
```

- Esto enumera el contenido del directorio *child-processes* en formato largo, junto con el tamaño del contenido en la parte superior. Sus resultados tendrán su propio usuario y grupo en lugar de *sammy*.
- Esto muestra que el programa *listFiles.js* ejecutó correctamente el comando *shell ls -lh*.

# ***Proceso secundario con 'execFile( )'***



- La diferencia principal entre las funciones `execFile()` y `exec()` es que el primer argumento de `execFile()` es ahora una ruta a un archivo ejecutable en vez de un comando.
- El resultado del archivo ejecutable se guarda en un búfer como `exec()`, al que accedemos a través de una función *callback* con los parámetros *error*, *stdout* y *stderr*.

# Proceso secundario con **'execFile( )'**



- Ahora requerimos el método *execFile* del módulo *child\_process*.
- Como observamos en el código, ahora el primer parámetro es la ruta, en este caso, de un script de bash.
- Luego, el código funciona de igual forma que con el comando *exec*.

```
const { execFile } = require('child_process');

execFile(__dirname + '/processNodejsImage.sh', (error, stdout, stderr) => {
  if (error) {
    console.error(`error: ${error.message}`);
    return;
  }

  if (stderr) {
    console.error(`stderr: ${stderr}`);
    return;
  }

  console.log(`stdout: \n${stdout}`);
});
```

# ***Proceso secundario con 'spawn( )'***



- La función *spawn()* ejecuta un comando en un proceso. Esta función devuelve datos a través de la API del flujo. Por tanto, para obtener el resultado del proceso secundario, debemos escuchar los eventos del flujo.
- Con *exec()* y *execFile()*, todos los datos procesados se guardan en la memoria de la computadora. Para cantidades de datos más grandes, esto puede degradar el rendimiento del sistema.
- En el caso de *spawn()*, con un flujo, los datos se procesan y transfieren en pequeños trozos. Por lo tanto, puede procesar una gran cantidad de datos sin usar demasiada memoria en un momento dado.

# Proceso secundario con **'spawn( )'**



- Requerimos el método *spawn* del módulo *child\_process*.
- El primer argumento de *spawn* es el comando ***find***.
- El segundo argumento es un *array* que contiene los argumentos para el comando ejecutado.
- Le estamos indicando a Node que ejecute el comando *find* con el argumento ***'.'***, lo que hace que el comando encuentre todos los activos en el directorio actual.

```
const { spawn } = require('child_process');  
const child = spawn('find', ['.']);
```

# Proceso secundario con **'spawn( )'**



- Los comandos pueden devolver datos en el flujo **stdout** o el flujo **stderr**.
- Puede añadir oyentes invocando el método **on()** de cada objeto de los flujos.
- El evento datos de los flujos nos proporciona el resultado de los comandos para ese flujo. Siempre que obtengamos datos sobre ese flujo, los registramos en la consola.

```
const { spawn } = require('child_process');  
  
const child = spawn('find', ['.']);  
  
child.stdout.on('data', data => {  
  console.log(`stdout: \n${data}`);  
});  
  
child.stderr.on('data', data => {  
  console.error(`stderr: ${data}`);  
});
```



# Proceso secundario con **'spawn( )'**



- Escuchamos los dos otros eventos: el evento **error** si el comando no se ejecuta o se interrumpe y el evento **close** para cuando el comando haya terminado la ejecución, cerrando así el flujo.
- En estos casos, se configura directo en la variable *child*.

```
const { spawn } = require('child_process');

const child = spawn('find', ['.']);

child.stdout.on('data', (data) => {
  console.log(`stdout:\n${data}`);
});

child.stderr.on('data', (data) => {
  console.error(`stderr: ${data}`);
});

child.on('error', (error) => {
  console.error(`error: ${error.message}`);
});

child.on('close', (code) => {
  console.log(`child process exited with code ${code}`);
});
```

# Proceso secundario con `'fork( )'`



- La función `fork()` es una variación de `spawn()` que permite la comunicación entre el proceso principal y el secundario.
- Además de recuperar datos desde el proceso secundario, un proceso principal puede enviar mensajes al proceso secundario en ejecución. Del mismo modo, el proceso secundario puede enviar mensajes al proceso principal.
- Si un servidor web está bloqueado, no puede procesar ninguna nueva solicitud entrante hasta que el código de bloqueo haya completado su ejecución. **Fork evita el bloqueo** corriendo el proceso secundario bloqueante en un hilo aparte.

# Proceso secundario con **'fork( )'**



```
const http = require('http')

const calculo = () => {
  let sum = 0
  for(let i=0; i<6e9; i++) {
    sum += i
  }
  return sum
}

let visitas = 0

const server = http.createServer()
server.on('request', (req,res) => {
  let { url } = req
  if(url == '/calcular') {
    const sum = calculo()
    res.end(`La suma es ${sum}`)
  }
  else if(url == '/') {
    res.end(`Ok ' + (++visitas)`)
  }
})

const PORT = 8080
server.listen(PORT, err => {
  if(err) throw new Error(`Error en servidor: ${err}`)
  console.log(`Servidor http escuchando en el puerto ${PORT}`)
})
```

- Creamos un servidor, que ejecuta la función *calculo()*.
- Se puede ver que se va a ejecutar de forma muy lenta, ya que el bucle de la función va a iterar hasta que el contador sea 6e9.
- Por lo tanto, este es un servidor bloqueante. Es decir, que todos los otros procesos o request se bloquean por el tiempo que tarda en ejecutarse la función *calcular*.

# Proceso secundario con **'fork( )'**



```
const http = require('http')
const { fork } = require('child_process')

let visitas = 0

const server = http.createServer()
server.on('request', (req,res) => {
  let { url } = req
  if(url == '/calcular') {
    //const sum = calculo()
    const computo = fork('./computo.js')
    computo.send('start')
    computo.on('message', sum => {
      res.end(`La suma es ${sum}`)
    })
  }
  else if(url == '/') {
    res.end('Ok ' + (++visitas))
  }
})

const PORT = 8080
server.listen(PORT, err => {
  if(err) throw new Error(`Error en servidor: ${err}`)
  console.log(`Servidor http escuchando en el puerto ${PORT}`)
})
```

- En este caso, tenemos un servidor no-bloqueante.
- La función *calcular*, ahora se encuentra en el archivo *computo.js*.
- Cuando se quiere acceder a la ruta */calcular* se crea un proceso secundario con *fork*.
- De esta forma, cualquier otro request se atenderá en forma correcta cuando se esté realizando la operación de cálculo.

# Proceso secundario con **'fork( )'**



En este ejemplo, tenemos un componente hijo, llamado *child.js* (primera imagen) el cual se va a ejecutar en un proceso secundario conectado con el proceso principal (segunda imagen).

En la *tercera imagen*, podemos observar el intercambio de mensajes entre el proceso hijo y el proceso padre al ejecutarlo en la terminal.

```
'child.js'

let contador = 0

process.on('message', msg => {
  console.log('Mensaje del padre: ', msg)

  setInterval(() => {
    process.send({ contador: contador++})
  }, 1000);
})
```

```
const { fork } = require('child_process')

const forked = fork('child.js')

forked.on('message', msg => {
  console.log('Mensaje del hijo ', msg)
})

console.log('Comienzo del programa Padre')
setTimeout(() => {
  forked.send({mensaje: 'Hola!'})
}, 2000)
```

```
Comienzo del programa Padre
Mensaje del padre: { mensaje: 'Hola!' }
Mensaje del hijo { contador: 0 }
Mensaje del hijo { contador: 1 }
Mensaje del hijo { contador: 2 }
Mensaje del hijo { contador: 3 }
```

# Proceso secundario con **'fork( )'**



Cuando trabajamos con Módulos de ES6, debemos considerar que la carga de archivos es no-bloqueante, por lo que debemos añadir un paso extra al cargar el nuevo proceso, que consiste en hacer que el proceso cargado envíe un mensaje indicando que ya está listo, y recién ahí podemos enviarle nosotros la indicación de que comience a procesar lo que le pasemos. Ejemplo:

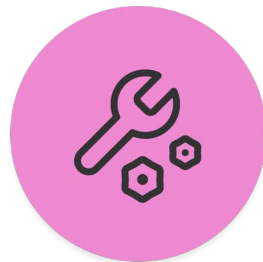
```
import { fork } from 'child_process'

const forked = fork('child.js')

forked.on('message', msg => {
  if (msg == 'listo') {
    forked.send('Hola, ')
  } else {
    console.log(`Mensaje del hijo: ${msg}`)
  }
})
```

```
// child.js
process.on('message', msg => {
  console.log(`mensaje del padre: ${msg}`)
  process.send('mundo!')
  process.exit()
})

process.send('listo')
```



# ***CHILD\_PROCESS CON FORK***

*Tiempo: 15 minutos*

# ***Child\_process con fork***

*Tiempo: 15 minutos*

Desafío  
generico



Realizar un servidor en express que contenga una ruta raíz '/' donde se represente la cantidad de visitas totales a este endpoint (no usar session).

Se implementará otra ruta '/calculo-bloq', que permita realizar una suma incremental de los números del 0 al 100000 con el siguiente algoritmo.

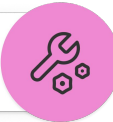
```
function sumar() {  
  let suma = 0  
  for(let i=0; i<5e9; i++) {  
    suma += i  
  }  
  return suma  
}
```



# ***Child\_process con fork***

*Tiempo: 15 minutos*

Desafío  
generico



Comprobar que al alcanzar esta ruta en una pestaña del navegador, el proceso queda en espera del resultado. Constatar que durante dicha espera, la ruta de visitas no responde hasta terminar este proceso.

Luego crear la ruta '/calculo-nobloq' que hará dicho cálculo forkeando el algoritmo en un child\_process, comprobando ahora que el request a esta ruta no bloquee la ruta de visitas. Asegurarse de que una vez finalizado el proceso, el worker se cierra correctamente.



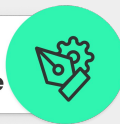
# ***USANDO EL OBJETO PROCESS***

# AGREGAR DOTENV

**Formato:** link a un repositorio en Github con el proyecto cargado.

**Sugerencia:** no incluir los node\_modules

Desafío  
entregable



## >> Consigna:

Sobre el proyecto del último desafío entregable, mover todas las claves y credenciales utilizadas a un archivo **.env**, y cargarlo mediante la librería **dotenv**.

La única configuración que no va a ser manejada con esta librería va a ser el puerto de escucha del servidor. Éste deberá ser leído de los argumento pasados por línea de comando, usando alguna librería (minimist o yargs). En el caso de no pasar este parámetro por línea de comandos, conectar por defecto al puerto 8080.

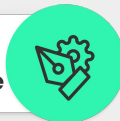
Observación: por el momento se puede dejar la elección de sesión y de persistencia explicitada en el código mismo. Más adelante haremos también parametrizable esta configuración.

# USANDO EL OBJETO PROCESS

**Formato:** link a un repositorio en Github con el proyecto cargado.

**Sugerencia:** no incluir los node\_modules

Desafío  
entregable



## >> Consigna:

Agregar una ruta '/info' que presente en una vista sencilla los siguientes datos:

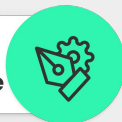
- Argumentos de entrada
- Nombre de la plataforma (sistema operativo)
- Versión de node.js
- Memoria total reservada (rss)
- Path de ejecución
- Process id
- Carpeta del proyecto

# USANDO EL OBJETO PROCESS

**Formato:** link a un repositorio en Github con el proyecto cargado.

**Sugerencia:** no incluir los node\_modules

Desafío  
entregable



## >> Consigna:

Agregar otra ruta '/api/randoms' que permita calcular un cantidad de números aleatorios en el rango del 1 al 1000 especificada por parámetros de consulta (query).

Por ej: /randoms?cant=20000.

Si dicho parámetro no se ingresa, calcular 100.000.000 números.

El dato devuelto al frontend será un objeto que contendrá como claves los números random generados junto a la cantidad de veces que salió cada uno. Esta ruta no será bloqueante (utilizar el método fork de child process). Comprobar el no bloqueo con una cantidad de 500.000.000 de randoms.

Observación: utilizar routers y apis separadas para esta funcionalidad.

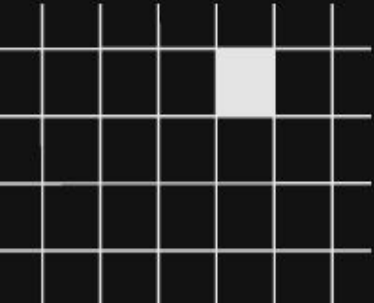
***¿PREGUNTAS?***





# ***¡MUCHAS GRACIAS!***

Resumen de lo visto en clase hoy:

- Global Process.
  - Child-process.
- 



***OPINA Y VALORA ESTA CLASE***



***#DEMOCRATIZANDO LA EDUCACIÓN***

***CODER HOUSE***