**Implementación de Clusters en aplicaciones de Node.js**

En este ejercicio, implementaremos *Clusters* en nuestra aplicación de Node.js. ¿Para qué sirven los *Clusters?*

Actualmente, la gran mayoría de computadoras y dispositivos móviles cuentan con procesadores multi-núcleos y muchos lenguajes de programación multihilo, asignan hilos a cada uno de los núcleos de los que disponga la computadora en la que se esté corriendo un determinado programa. Pero en Node.js sólo disponemos de un hilo “maestro”. Bien, para poder aprovechar sistemas multi-núcleo, podremos implementar *Clusters* y así incrementar los recursos que nuestro único hilo en Node.js. Y al igual que en anterior ejercicio sólo tendremos que editar el archivo *app.js:*

*app.js*

"use strict";

**const** cluster = require('cluster');

**const** cpuCount = require('os').cpus().length;

*//Revisamos si el proceso actual es el proceso 'master' o maestro.*

if (cluster.isMaster) {

console.log(`Master ${process.pid} está ejecutándose ;)`);

console.log(`Núcleos del equipo actual: `, cpuCount);

*// Creamos un proceso 'worker' por cada núcleo.*

for (**let** i = 0; i < cpuCount; i += 1) {

cluster.fork();

}

*// Callback que revisa si un worker muere :(*

cluster.on('exit', (worker) **=>** {

*// En caso de perder un proceso 'worker'*

console.log(`Worker ${worker.id} died :(`);

*// creamos uno nuevo.*

cluster.fork();

});

} else {

**const** express = require('express');

**const** app = express();

**const** bodyParser = require('body-parser');

app.use(bodyParser.json());

**const** cors = require('cors');

app.use(cors());

**const** env = require('dotenv');

env.config();

**const** pendiente = require('./pendiente/pendiente.routes');

app.use('/pendiente', pendiente);

**const** portExpress = process.env.EXPRESS\_PORT;

**const** hostExpress = process.env.EXPRESS\_HOST;

app.listen(portExpress, hostExpress, () **=>** {

console.log(`El worker ${cluster.worker.id} está listo para recibir peticiones :)!`);

});

}

Bien, lo primero que haremos es requerir el módulo interno de *Node.js* llamado *cluster* y guardaremos su referencia en una constante con el mismo nombre*:*

**const** cluster = require('cluster');

Después, con el módulo interno de *Node.js* llamado ***os,*** preguntaremos con cuántos núcleos cuenta el equipo actual y lo que nos retorne lo guardarmos en un contador llamado *cpuCount:*

**const** cpuCount = require('os').cpus().length;

Después con una sentencia ***if*** verificaremos si el proceso actual es el proceso *master (*o maestro, en español*):*

if (cluster.isMaster) {...}

El proceso *master* es el proceso principal y es el proceso desde el cual podremos crear más proceso hijos llamados *workers*.

Si resulta que estamos en el proceso *master* *Node* ejecutará la siguiente porción de código, en la cual simplemente imprimimos el proceso *master* en la consola y la cantidad de núcleos que tiene el equipo actual:

console.log(`Master ${process.pid} está ejecutándose ;)`);

console.log(`Núcleos del equipo actual: `, cpuCount);

Después, se ejecutará un ciclo ***for*** con el cual se creará un proceso *hijo* (*worker)* por cada núcleo con el que cuente el equipo en el que se esté ejecutando la aplicación:

*// Creamos un proceso 'worker' por cada núcleo.*

for (**let** i = 0; i < cpuCount; i += 1) {

cluster.fork();

}

Enseguida, estaremos revisando si muere un *worker* con “cluster.on('exit', (worker) **=>** {...})*”* y por cada *worker* que muera, imprimiremos el ID del worker en cuestión en consola y crearemos uno nuevo para reemplazarlo con “cluster.fork();”.

*// Evento que revisa si un worker muere :(*

cluster.on('exit', (worker) **=>** {

*// En caso de perder un proceso 'worker'*

console.log(`Worker ${worker.id} died :(`);

*// creamos uno nuevo.*

cluster.fork();

});

Finalmente, en caso de que “if (cluster.isMaster) {...}*”* no se cumpla, pasaremos a su sentencia ***else*** en la cual definiremos la lógica de nuestro servidor como lo hemos estado haciendo a lo largo de los ejercicios:

} else {

**const** express = require('express');

**const** app = express();

**const** bodyParser = require('body-parser');

app.use(bodyParser.json());

**const** cors = require('cors');

app.use(cors());

**const** env = require('dotenv');

env.config();

**const** pendiente = require('./pendiente/pendiente.routes');

app.use('/pendiente', pendiente);

**const** portExpress = process.env.EXPRESS\_PORT;

**const** hostExpress = process.env.EXPRESS\_HOST;

app.listen(portExpress, hostExpress, () **=>** {

console.log(`El worker ${cluster.worker.id} está listo para recibir peticiones :)!`);

});

}

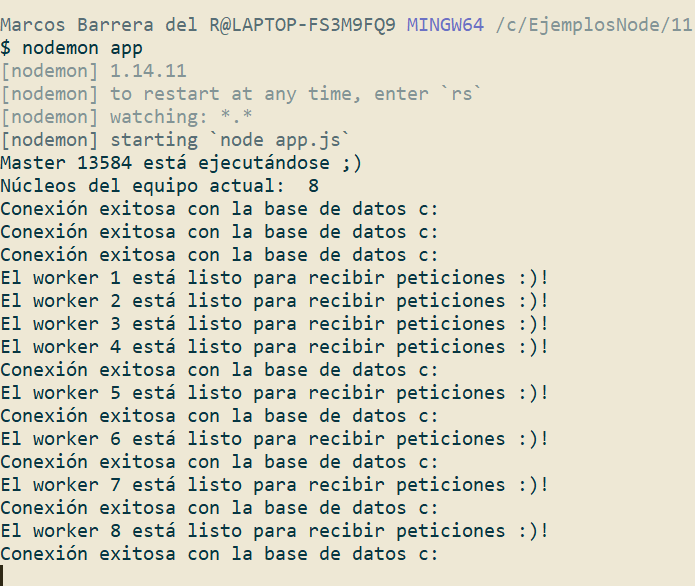
Y como habrás notado, cambiamos el mensaje de app.listen*.* Ahora imprimimos un mensaje que indica que un determinado *worker* está a la escucha de peticiones:

app.listen(portExpress, hostExpress, () **=>** {

console.log(`El worker ${cluster.worker.id} está listo para recibir peticiones :)!`);

});

Si ejecutamos nuestra aplicación con ***nodemon app***, veremos algo similar a la siguiente imagen:



Lo que indica que tenemos ocho *workers* (uno por cada núcleo) a la espera de peticiones. Y también veremos que el mensaje de “*Conexión exitosa con la base de datos c:*” se imprimió ocho veces, lo que no sugiere que el código de la sentencia ***else*** se ejecutó ocho veces, cada uno de los *workers* utilizó la conexión a *MySQL,* y por lo tanto el mensaje de la conexión exitosa se imprimió ocho veces.

Además, nos gustaría mencionar otra cualidad que tiene el utilizar *Clusters* en Node.js, y dicha cualidad es que si por alguna razón, alguna parte de nuestro código arroja un error y cierra el proceso actual, automáticamente el evento de “cluster.on('exit', (worker) **=>** {...});*”* se ejecutará y creará un *worker* nuevo, sin que nuestro servidor de *Node.js* deje de estar disponible. Esto nos otorga una ventaja grandísima, porque si no estuviésemos utilizamos *Clusters*, si dicho error surgiera y cerrara el proceso actual, *Node* dejaría de ejecutarse y nuestro servidor quedaría fuera de servicio hasta que manualmente lo restaurásemos.

*Por* último, dejaremos los archivos restantes de la aplicación para que los tengas cómo referencia:

*.env*

**#Las líneas que empiezan con un '#', son tratadas**

**#como comentario y por lo tanto, son ignoradas.**

**#NO SE RECOMIENDA SUBIR EL ARCHIVO .env A REPOSITORIOS**

**#DE GIT, PERO POR CUESTIONES PRÁCTICAS, LO SUBIREMOS ESTA VEZ**

**#Configuración de MySQL**

**DB\_HOST=127.0.0.1**

**DB\_USER=root**

**DB\_PASSWORD=qwerty1234**

**DB\_PORT=3306**

**DB\_DATABASE=db\_pendientes**

**#Configuración de nuestro servidor**

**EXPRESS\_PORT=3000**

**EXPRESS\_HOST=127.0.0.1**

*pendiente/pendiente.model.js*

**const** conexion = require('../db-config/mysql-connection');

**let** Pendiente = { };

Pendiente.obtenerTodos = ( res, cb ) **=>** {

if ( conexion ) {

conexion.query('SELECT pendiente\_id, descripcion, estado FROM pendientes', (error, resultados) **=>** {

if (error)

return cb( error, res );

return cb( null, res, resultados );

})

} else

return cb('Hubo un error con la conexión a MySQL :(', res);

}

Pendiente.contador = ( res, cb ) **=>** {

if ( conexion ) {

conexion.query('SELECT COUNT (pendiente\_id) AS numero\_de\_registros FROM pendientes', (error, contador) **=>** {

if ( error )

return cb( error, res );

return cb( null, res, contador );

})

} else

return cb('Hubo un error con la conexión a MySQL :(', res);

}

Pendiente.siguienteId = ( res, cb ) **=>** {

if ( conexion ) {

conexion.query(`SELECT \`AUTO\_INCREMENT\` AS siguiente\_id\_autoincrementable

FROM INFORMATION\_SCHEMA.TABLES

WHERE TABLE\_SCHEMA = '${process.env.DB\_DATABASE}'

AND TABLE\_NAME = 'pendientes';`, (error, siguienteId) **=>** {

if (error)

return cb( error, res );

return cb( null, res, siguienteId );

})

} else

return cb('Hubo con error con la conexión a MySQL :(', res);

}

Pendiente.encontrarPorId = ( idPendiente, res, cb ) **=>** {

if ( conexion ) {

conexion.query(`SELECT pendiente\_id, descripcion, estado

FROM pendientes

WHERE pendiente\_id = ?`, [idPendiente], (error, pendiente) **=>** {

if (error)

return cb( error, res );

return cb( null, res, pendiente );

})

} else

return cb('Hubo con error con la conexión a MySQL :(', res);

}

Pendiente.insertar = ( nuevoPendiente, res, cb ) **=>** {

if ( conexion ) {

conexion.query('INSERT INTO pendientes SET ?', [nuevoPendiente], (error, respuesta) **=>** {

if ( error )

return cb( error, res );

return cb( null, res, respuesta )

});

} else

return cb('Hubo con error con la conexión a MySQL :(', res);

}

Pendiente.actualizar = ( pendientePorEditar, res, cb ) **=>** {

if ( conexion ) {

conexion.query(`UPDATE pendientes

SET descripcion = ?, estado = ?

WHERE pendiente\_id = ?`,

[pendientePorEditar.descripcion,

pendientePorEditar.estado,

pendientePorEditar.pendiente\_id], (error, respuesta) **=>** {

if ( error )

return cb( error, res );

return cb( null, res, respuesta )

});

} else

return cb('Hubo con error con la conexión a MySQL :(', res)

}

Pendiente.eliminar = ( idPendiente, res, cb ) **=>** {

if ( conexion ) {

conexion.query(`DELETE FROM pendientes

WHERE pendiente\_id = ?`, [idPendiente], (error, respuesta) **=>** {

*//Si hay un error, le respondemos al cliente con el error.*

if (error)

return cb( error, res );

return cb( null, res, respuesta );

})

} else

return cb('Hubo un error con la conexión a MySQL :(', res);

}

Pendiente.responderAlCliente = ( error, res, datos ) **=>** {

if ( error )

res.status(500).json(error);

else

res.status(200).json(datos);

}

module.exports = Pendiente;

*db-config/mysql-connection.js*

**const** mysql = require('mysql');

**const** conexion = mysql.createConnection({

host: process.env.DB\_HOST,

user: process.env.DB\_USER,

password: process.env.DB\_PASSWORD,

port: process.env.DB\_PORT,

database: process.env.DB\_DATABASE

});

conexion.connect(err **=>** {

if (err) {

console.log('Error trying to connect with Data Base: ' + err.stack);

throw err;

}

console.log("Conexión exitosa con la base de datos c:")

});

module.exports = conexion;

*pendiente/pendientes.routes.js*

**const** express = require('express');

**const** router = express.Router();

*//Importamos el modelo de Pendiente.*

**const** Pendiente = require('./pendiente.model');

*//Consulta general de todos los pendientes en la base de datos.*

router

.get('/', (req, res) **=>** {

return Pendiente.obtenerTodos( res, Pendiente.responderAlCliente );

})

*//Consultar cuando registros hay en la tabla de pendientes.*

.get('/count', (req, res) **=>** {

return Pendiente.contador( res, Pendiente.responderAlCliente );

})

*//Consultar cuando registros hay en la tabla de pendientes.*

.get('/siguienteIdAutoIncrementable', (req, res) **=>** {

return Pendiente.siguienteId( res, Pendiente.responderAlCliente );

})

*//Consultar un pendiente por id.*

.get('/:idPendiente', (req, res) **=>** {

**const** idPendiente = req.params.idPendiente;

return Pendiente.encontrarPorId( idPendiente, res, Pendiente.responderAlCliente );

})

*//Agregar un nuevo pendiente a la base de datos.*

.post('/', (req, res) **=>** {

**const** nuevoPendiente = {

pendiente\_id: null,

descripcion: req.body.descripcion,

estado: req.body.estado

}

return Pendiente.insertar( nuevoPendiente, res, Pendiente.responderAlCliente );

})

*//Modifica un pendiente existente en la base de datos.*

.put('/:idPendiente', (req, res) **=>** {

**const** idPendiente = req.params.idPendiente;

**const** pendientePorEditar = {

pendiente\_id: idPendiente,

descripcion: req.body.descripcion,

estado: req.body.estado

}

return Pendiente.actualizar( pendientePorEditar, res, Pendiente.responderAlCliente );

})

*//Elimina un pendiente existente en la base de datos.*

.delete('/:idPendiente', (req, res) **=>** {

**const** idPendiente = req.params.idPendiente;

return Pendiente.eliminar( idPendiente, res, Pendiente.responderAlCliente );

})

module.exports = router;