

Desafío 13 - Logs, debug, profiling

- Incorporar al proyecto de servidor de trabajo la compresión gzip.
Verificar sobre la ruta /info con y sin compresión, la diferencia de cantidad de bytes devueltos en un caso y otro.

Ruta /info sin compresión:

0 B / 1.9 kB transferred	0 B / 1.1 kB resources	Finish: 314 ms	DOMContentLoaded: 332 ms	Load: 330 ms
What's New ✕				

Ruta /info con compresión:

0 B / 1.1 kB transferred	0 B / 1.1 kB resources	Finish: 48 ms	DOMContentLoaded: 54 ms	Load: 53 ms
What's New ✕				

Análisis: Efectivamente cuando se aplica compresión se verifica una gran diferencia en la cantidad de bytes y en los milisegundos.

- Vamos a trabajar sobre la ruta '/info', en modo fork, agregando ó extrayendo un console.log de la información colectada antes de devolverla al cliente. Además desactivaremos el child_process de la ruta '/randoms'

Para ambas condiciones (con o sin console.log) en la ruta '/info' OBTENER:

- 1) El perfilamiento del servidor, realizando el test con --prof de node.js. Analizar los resultados obtenidos luego de procesarlos con --prof-process.

Se realizó Artillery emulando 50 conexiones simultaneas con 20 request por cada una y se verifica el siguiente resultado de los logs:

[Summary]:			
ticks	total	nonlib	name
11	0.4%	100.0%	JavaScript
0	0.0%	0.0%	C++
7	0.2%	63.6%	GC
2849	99.6%		Shared libraries

Este resultado es CON CONSOLE.LOG

Cuando index.js posee console.log() hay muchos mas ticks que cuando no los posee. La aplicación CON console.log() tardó 2849 ciclos de reloj en ejecutarse.

[Summary]:			
ticks	total	nonlib	name
5	0.2%	100.0%	JavaScript
0	0.0%	0.0%	C++
6	0.3%	120.0%	GC
2193	99.8%		Shared libraries

Este resultado es SIN CONSOLE.LOG

Un tic es como un ciclo de reloj utilizado por un proceso de nodo. Entonces, en teoría, la aplicación SIN console.log() tardó 2193 ciclos de reloj en ejecutarse.

La diferencia de ticks es de 650, no es mucha diferencia pero si apuntamos a una mejor performance hay que tener en cuenta que los console.log() afectan a la misma.

Utilizaremos como test de carga Artillery en línea de comandos, emulando 50 conexiones concurrentes con 20 request por cada una. Extraer un reporte con los resultados en archivo de texto.

Con console.log	Sin console.log
Phase started: unnamed (index: 0, duration: 1s) 01:26:07(-0300) Phase completed: unnamed (index: 0, duration: 1s) 01:26:08(-0300) All VUs finished. Total time: 10 seconds ----- Summary report @ 01:26:15(-0300) ----- http.codes.200: 1000 http.request_rate: 142/sec http.requests: 1000 http.response_time: min: 5 max: 274 median: 133 p95: 179.5 p99: 186.8 http.responses: 1000 vusers.completed: 50 vusers.created: 50 vusers.created_by_name.0: 50 vusers.failed: 0 vusers.session_length: min: 1077.2 max: 2797.7 median: 2618.1 p95: 2780 p99: 2780	Phase started: unnamed (index: 0, duration: 1s) 01:36:12(-0300) Phase completed: unnamed (index: 0, duration: 1s) 01:36:13(-0300) All VUs finished. Total time: 5 seconds ----- Summary report @ 01:36:15(-0300) ----- http.codes.200: 1000 http.request_rate: 365/sec http.requests: 1000 http.response_time: min: 2 max: 54 median: 22.9 p95: 34.1 p99: 44.3 http.responses: 1000 vusers.completed: 50 vusers.created: 50 vusers.created_by_name.0: 50 vusers.failed: 0 vusers.session_length: min: 169.9 max: 599 median: 487.9 p95: 596 p99: 596

- Luego utilizaremos Autocannon en línea de comandos, emulando 100 conexiones concurrentes realizadas en un tiempo de 20 segundos. Extraer un reporte con los resultados (puede ser un print screen de la consola)

2) El perfilamiento del servidor con el modo inspector de node.js --inspect. Revisar el tiempo de los procesos menos performantes sobre el archivo fuente de inspección.

CON CONSOLE.LOG()

Self Time	Total Time	Function
17164.6 ms	17164.6 ms	(idle)
1131.1 ms 24.58 %	2456.1 ms 53.38 %	▶ consoleCall

SIN CONSOLE.LOG()

Self Time	Total Time	Function
13497.4 ms	13497.4 ms	(idle)
131.7 ms 8.03 %	131.7 ms 8.03 %	▶ getCPUs

CON CONSOLE.LOG()

232		////////////////////////////////////
233		app.get('/info', (req, res) => {
234	1.4 ms	const argEntrada = process.argv[2];
235	1.3 ms	const plataforma = process.title;
236	0.2 ms	const versionNode = process.version;
237	0.6 ms	const memoriaTotal = process.memoryUsage().heapTotal;
238		const pathEjecucion = process.execPath;
239	0.6 ms	const processId = process.pid;
240	0.2 ms	const carpetaProyecto = process.cwd();
241	1.7 ms	const procesadoresNum = require('os').cpus().length;
242	0.9 ms	const array = [argEntrada, plataforma, versionNode, memoriaTotal, pathEjecucion, processId, carpetaProyecto, procesadoresNum];
243	4.1 ms	console.log(array);
244	15.8 ms	res.render('pages/info/info', {
245		argumentos: argEntrada,
246		plataforma: plataforma,
247		versionNode: versionNode,
248	0.1 ms	memoriaTotal: memoriaTotal,
249		pathEjecucion: pathEjecucion,
250		processId: processId,
251		carpetaProyecto: carpetaProyecto,
252	0.2 ms	procesadoresNum: procesadoresNum
253		});
254		});

SIN CONSOLE.LOG()

232		////////////////////////////////////
233		app.get('/info', (req, res) => {
234	0.7 ms	const argEntrada = process.argv[2];
235	2.1 ms	const plataforma = process.title;
236	0.3 ms	const versionNode = process.version;
237	0.8 ms	const memoriaTotal = process.memoryUsage().heapTotal;
238	0.2 ms	const pathEjecucion = process.execPath;
239		const processId = process.pid;
240	0.2 ms	const carpetaProyecto = process.cwd();
241	2.1 ms	const procesadoresNum = require('os').cpus().length;
242	0.4 ms	const array = [argEntrada, plataforma, versionNode, memoriaTotal, pathEjecucion, processId, carpetaProyecto, procesadoresNum];
243		//console.log(array);
244	5.7 ms	res.render('pages/info/info', {
245	0.1 ms	argumentos: argEntrada,
246		plataforma: plataforma,
247		versionNode: versionNode,
248		memoriaTotal: memoriaTotal,
249		pathEjecucion: pathEjecucion,
250		processId: processId,
251		carpetaProyecto: carpetaProyecto,
252	0.1 ms	procesadoresNum: procesadoresNum
253		});
254		});

- 3) El diagrama de flama con 0x, emulando la carga con Autocannon con los mismos parámetros anteriores.

SIN CONSOLE.LOG()

PS C:\Users\S4351\Desktop\CODER - BACKEND\Desafío 14 -Loggers, gzip y análisis de performance> node --inspect index.js
 Debugger listening on ws://127.0.0.1:9229/f2d6506e-eb64-4dc2-afdd-dee34bc4126c
 For help, see: https://nodejs.org/en/docs/inspector

Latency	15 ms	37 ms	72 ms	80 ms	39.34 ms	14.9 ms	212 ms
---------	-------	-------	-------	-------	----------	---------	--------

Stat	1%	2.5%	50%	97.5%	Avg	Stdev	Min
Req/Sec	689	689	1295	1467	1254.2	207.49	689
Bytes/Sec	908 kB	908 kB	1.71 MB	1.93 MB	1.65 MB	274 kB	908 kB

Req/Bytes counts sampled once per second.
 # of samples: 20
 25k requests in 20.04s, 33.1 MB read

CON CONSOLE.LOG()

PS C:\Users\S4351\Desktop\CODER - BACKEND\Desafío 14 -Loggers, gzip y análisis de performance> node benchmark.js
 Running all benchmarks in parallel ...
 Running 20s test @ http://localhost:8080/info
 50 connections

Stat	2.5%	50%	97.5%	99%	Avg	Stdev	Max
Latency	28 ms	90 ms	140 ms	157 ms	87.61 ms	28.24 ms	279 ms

Stat	1%	2.5%	50%	97.5%	Avg	Stdev	Min
Req/Sec	448	448	558	656	566.85	59.68	448
Bytes/Sec	591 kB	591 kB	736 kB	865 kB	747 kB	78.6 kB	590 kB

Req/Bytes counts sampled once per second.
 # of samples: 20
 11k requests in 20.05s, 14.9 MB read

[illegible]