

Informe de gzip y profiling

1. Resultados de la compresión con GZIP

SIN GZIP

Name	Status	Type	Initiator	Size	Time	Waterfall
info	200	docu...	Other	2.0 kB	426 ms	
bootstrap.min.css	200	styles...	info	(mem...	0 ms	
normalizr.browser.min.js	200	script	info	(mem...	0 ms	
bootstrap.bundle.min.js	200	script	info	(mem...	0 ms	
styles.css	304	styles...	info	265 B	8 ms	

CON GZIP

Name	Status	Type	Initiator	Size	Time	Waterfall
info	200	docu...	Other	1.1 kB	412 ms	
bootstrap.min.css	200	styles...	info	(mem...	0 ms	
normalizr.browser.min.js	200	script	info	(mem...	0 ms	
bootstrap.bundle.min.js	200	script	info	(mem...	0 ms	
styles.css	304	styles...	info	265 B	9 ms	
socket.io.js	304	script	info	113 B	9 ms	
?EIO=4&transport=polling...	200	xhr	polling-xhr.js...	261 B	11 ms	
?EIO=4&transport=polling...	200	xhr	polling-xhr.js...	149 B	6 ms	

Compresión: 0.9 kB

2. Profiling con node y Artillery

CON CONSOLE.LOG()

```

=====
Summary report @ 15:17:58(-0300)
=====

http.codes.200: ..... 1000
http.request_rate: ..... 14/sec
http.requests: ..... 1000
http.response_time:
  min: ..... 7
  max: ..... 4550
  median: ..... 214.9
  p95: ..... 1022.7
  p99: ..... 2725
  
```

SIN CONSOLE.LOG()

```

=====
Summary report @ 15:22:02(-0300)
=====

http.codes.200: ..... 1000
http.request_rate: ..... 14/sec
http.requests: ..... 1000
http.response_time:
  min: ..... 5
  max: ..... 4631
  median: ..... 219.2
  p95: ..... 1130.2
  p99: ..... 2836.2
http.responses: ..... 1000

```

- Llama la atención que el promedio de tiempo de respuesta sea más alto para los requests que no utilizaron console.log

3. Profiling con node inspect y chrome

Connection Console Sources Memory Profiler			
Heavy (Bottom Up) ▾ 🔍 ✕ ↺			
Profiles	Self Time	Total Time	Function
CPU PROFILES	25521.6 ms	25521.6 ms	(idle)
Profile 1 Save	152.0 ms 7.79 %	200.0 ms 10.25 %	▶ consoleCall
	114.4 ms 5.86 %	114.4 ms 5.86 %	(garbage collector)
	95.0 ms 4.87 %	95.0 ms 4.87 %	▶ getCPUs
	70.1 ms 3.59 %	70.1 ms 3.59 %	(program)
	29.5 ms 1.51 %	86.9 ms 4.45 %	▶ deserializeObject deserializer.js:65
	26.3 ms 1.35 %	26.3 ms 1.35 %	▶ writeUtf8String
	24.4 ms 1.25 %	119.1 ms 6.10 %	▶ SourceNode_walk source-node.js:221
	23.4 ms 1.20 %	23.4 ms 1.20 %	▶ open
	21.9 ms 1.12 %	68.3 ms 3.50 %	▶ parse parser.js:269
	21.7 ms 1.11 %	21.7 ms 1.11 %	▶ memoryUsage
	21.4 ms 1.10 %	28.2 ms 1.45 %	▶ SourceNode_add source-node.js:172
	20.0 ms 1.03 %	21.4 ms 1.10 %	▶ writeBuffer
	19.0 ms 0.97 %	264.8 ms 13.58 %	▶ compile javascript-compiler.js:73
	18.6 ms 0.95 %	26.1 ms 1.34 %	▶ next parser.js:478
	16.5 ms 0.84 %	819.6 ms 42.02 %	▶ callbackTrampoline node:internal/async_hooks:118
	16.5 ms 0.84 %	51.8 ms 2.66 %	▶ createFunctionContext javascript-compiler.js:216
	15.8 ms 0.81 %	15.8 ms 0.81 %	▶ quotedString code-gen.js:118
	15.8 ms 0.81 %	370.4 ms 18.99 %	▶ compileInput compiler.js:507
	15.5 ms 0.79 %	91.4 ms 4.69 %	▶ wrap code-gen.js:101
	14.2 ms 0.73 %	23.8 ms 1.22 %	▶ nextTick node:internal/p...ask_queues:104
	14.2 ms 0.73 %	14.2 ms 0.73 %	▶ writev
	13.2 ms 0.68 %	2324.4 ms 119.16 %	▶ next index.js:176

- Tiene sentido que el proceso que más haya demandado al servidor sea la impresión por consola de cada request (por parte de log4js)

4. Impresión por consola de Autocannon

```
> loggers@1.0.0 test  
> node ac_test.js
```

```
Running 20s test @ http://localhost:8080/info  
100 connections
```

Stat	2.5%	50%	97.5%	99%	Avg	Stdev	Max
Latency	10 ms	6470 ms	9625 ms	9633 ms	4646.99 ms	3638.72 ms	9636 ms

Stat	1%	2.5%	50%	97.5%	Avg	Stdev	Min
Req/Sec	0	0	4	46	7.75	10.72	1
Bytes/Sec	0 B	0 B	8.97 kB	103 kB	17.4 kB	24.1 kB	2.24 kB

```
Req/Bytes counts sampled once per second.  
# of samples: 20
```

```
310 requests in 20.13s, 348 kB read  
55 errors (55 timeouts)
```

El informe de 0x está disponible en la carpeta 0x_profiling ubicada en el root del proyecto