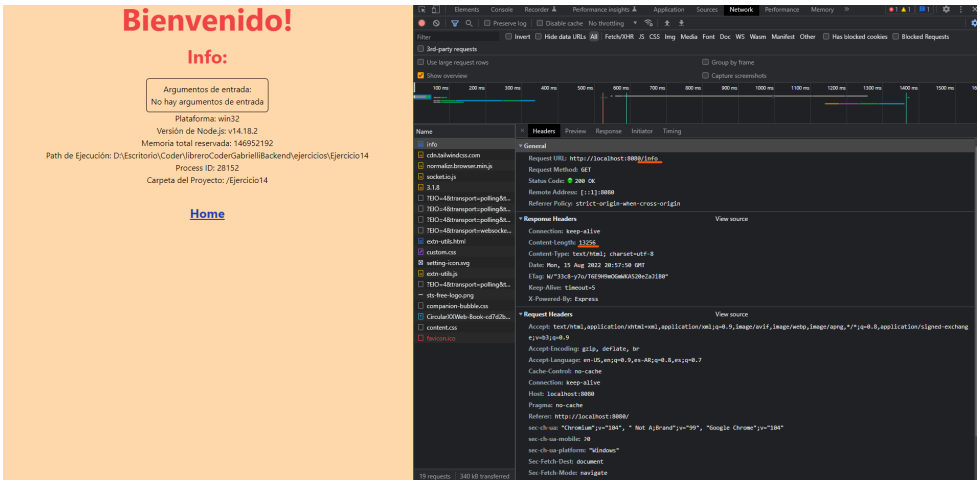
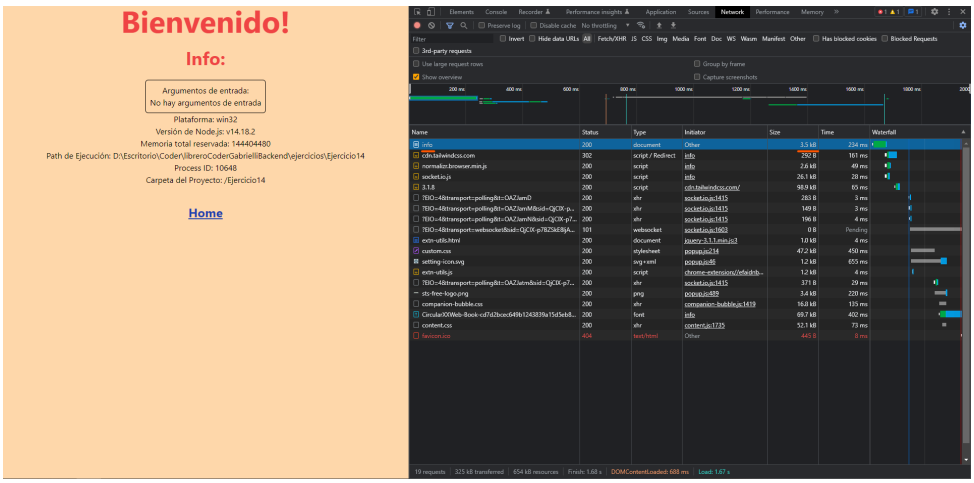
Análisis de Performance

Respuesta **Pre-Compresión**:

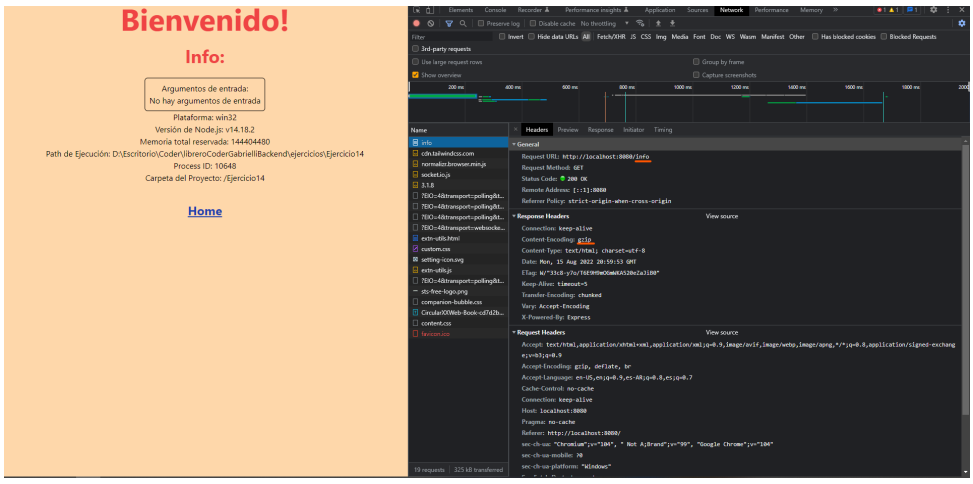


13256kB

Respuesta **Post-Compresión**:



3,5kb



Diferencia en kB: 13,252,5

Resultados de profiling

1) **Resultados profiling con node --prof:** El proceso que no utiliza console.log() tiene algunos ticks más de duración, pero utiliza menos librerías compartidas que el que si lo utiliza.
2) **Resultados profiling con Autocannon**
    a. **Autocannon sin console.log():**

```
voice@DESKTOP-059N20J MINGW64 /d/Escritorio/Coder/libreroCoderGabrielliBackend/ejercic
ios/ejercicio14 (master)
$ autocannon http://localhost:8080/info -d 20 -c 100
Running 20s test @ http://localhost:8080/info
100 connections
```

Stat	2.5%	50%	97.5%	99%	Avg	Stdev	Max
Latency	399 ms	440 ms	531 ms	561 ms	450.37 ms	36.07 ms	599 ms

Stat	1%	2.5%	50%	97.5%	Avg	Stdev	Min
Req/Sec	124	124	215	272	219.85	32.15	124
Bytes/Sec	1.67 MB	1.67 MB	2.9 MB	3.67 MB	2.97 MB	434 kB	1.67 MB

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

4k requests in 20.14s, 59.3 MB read
```

    b. **Autocannon con console.log():**

```
voice@DESKTOP-059N20J MINGW64 /d/Escritorio/Coder/libreroCoderGabrielliBackend/ejercic
ios/ejercicio14 (master)
$ autocannon http://localhost:8080/info -d 20 -c 100
Running 20s test @ http://localhost:8080/info
100 connections
```

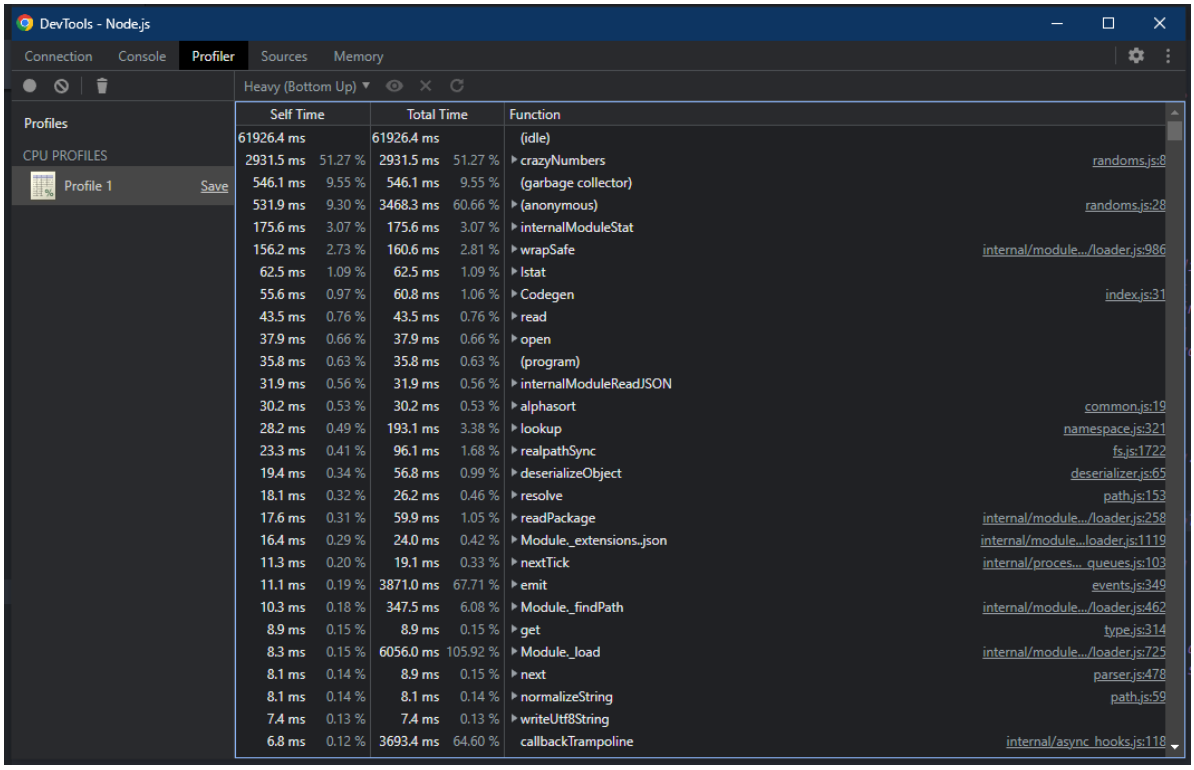
Stat	2.5%	50%	97.5%	99%	Avg	Stdev	Max
Latency	605 ms	667 ms	939 ms	990 ms	694.73 ms	78.23 ms	1026 ms

Stat	1%	2.5%	50%	97.5%	Avg	Stdev	Min
Req/Sec	90	90	114	198	144.1	40.6	90
Bytes/Sec	1.21 MB	1.21 MB	1.54 MB	2.67 MB	1.94 MB	547 kB	1.21 MB

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

3k requests in 20.21s, 38.9 MB read
```

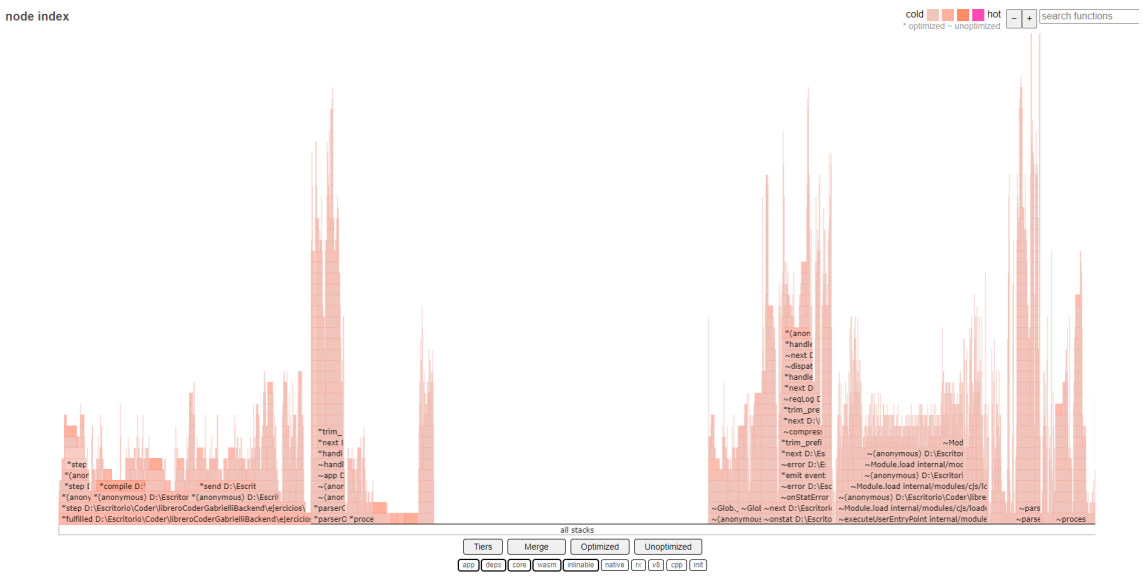
3) Profiling con Node –inspect y Autocannon:



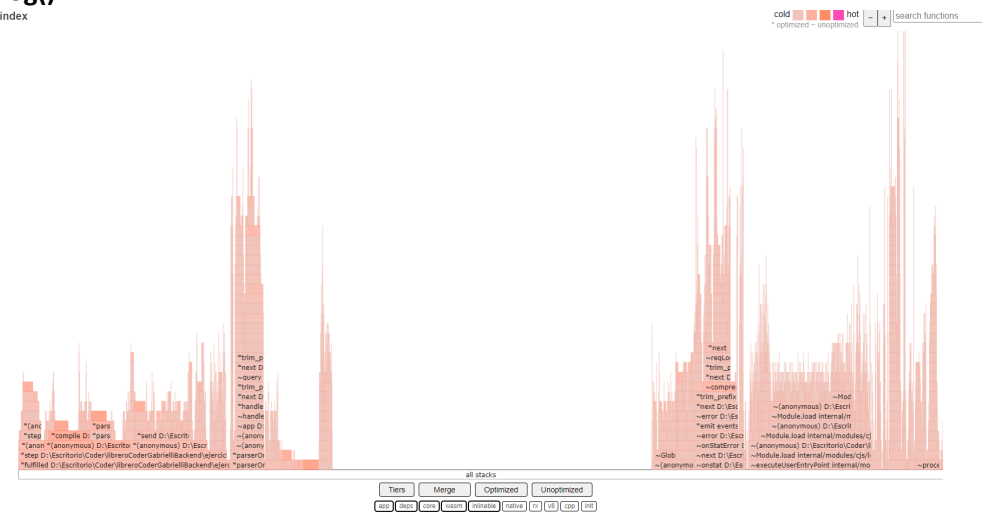
Self Time		Total Time		Function	
61926.4 ms		61926.4 ms		(idle)	
2931.5 ms	51.27 %	2931.5 ms	51.27 %	crazyNumbers	randoms.js:8
546.1 ms	9.55 %	546.1 ms	9.55 %	(garbage collector)	
531.9 ms	9.30 %	3468.3 ms	60.66 %	(anonymous)	randoms.js:28
175.6 ms	3.07 %	175.6 ms	3.07 %	internalModuleStat	
156.2 ms	2.73 %	160.6 ms	2.81 %	wrapSafe	internal/module.../loader.js:986
62.5 ms	1.09 %	62.5 ms	1.09 %	lstat	
55.6 ms	0.97 %	60.8 ms	1.06 %	Codegen	index.js:31
43.5 ms	0.76 %	43.5 ms	0.76 %	read	
37.9 ms	0.66 %	37.9 ms	0.66 %	open	
35.8 ms	0.63 %	35.8 ms	0.63 %	(program)	
31.9 ms	0.56 %	31.9 ms	0.56 %	internalModuleReadJSON	
30.2 ms	0.53 %	30.2 ms	0.53 %	alphasort	common.js:19
28.2 ms	0.49 %	193.1 ms	3.38 %	lookup	namespace.js:321
23.3 ms	0.41 %	96.1 ms	1.68 %	realpathSync	fs.js:1722
19.4 ms	0.34 %	56.8 ms	0.99 %	deserializeObject	deserializer.js:65
18.1 ms	0.32 %	26.2 ms	0.46 %	resolve	path.js:153
17.6 ms	0.31 %	59.9 ms	1.05 %	readPackage	internal/module.../loader.js:258
16.4 ms	0.29 %	24.0 ms	0.42 %	Module._extensions..json	internal/module...loader.js:1119
11.3 ms	0.20 %	19.1 ms	0.33 %	nextTick	internal/proces... queues.js:103
11.1 ms	0.19 %	3871.0 ms	67.71 %	emit	events.js:349
10.3 ms	0.18 %	347.5 ms	6.08 %	Module._findPath	internal/module.../loader.js:462
8.9 ms	0.15 %	8.9 ms	0.15 %	get	type.js:314
8.3 ms	0.15 %	6056.0 ms	105.92 %	Module._load	internal/module.../loader.js:725
8.1 ms	0.14 %	8.9 ms	0.15 %	next	parser.js:478
8.1 ms	0.14 %	8.1 ms	0.14 %	normalizeString	path.js:59
7.4 ms	0.13 %	7.4 ms	0.13 %	writeUtf8String	
6.8 ms	0.12 %	3693.4 ms	64.60 %	callbackTrampoline	internal/async_hooks.js:118

4) Profiling con 0x

a. Con console.log():



b. **Sin console.log():**



5) Tests de carga con Artillery:

a. **Con console.log():**

```
Running scenarios...
Phase started: unnamed (index: 0, duration: 1s) 20:26:48(-0300)

Phase completed: unnamed (index: 0, duration: 1s) 20:26:49(-0300)

All VUs finished. Total time: 15 seconds

--------------------------------
Summary report @ 20:27:00(-0300)
--------------------------------

http.codes.200: ..................................................... 1000
http.request_rate: ..................................................... 57/sec
http.requests: ..................................................... 1000
http.response_time:
min: ..................................................... 48
max: ..................................................... 529
median: ..................................................... 383.8
p95: ..................................................... 468.8
p99: ..................................................... 528.6
http.responses: ..................................................... 1000
vusers.completed: ..................................................... 50
vusers.created: ..................................................... 50
vusers.created_by_name.0: ..................................................... 50
vusers.failed: ..................................................... 0
vusers.session_length:
min: ..................................................... 7337.8
max: ..................................................... 7817.7
median: ..................................................... 7709.8
p95: ..................................................... 7865.6
p99: ..................................................... 7865.6
```

b. **Sin console.log():**

```
Running scenarios...
Phase started: unnamed (index: 0, duration: 1s) 20:27:39(-0300)

Phase completed: unnamed (index: 0, duration: 1s) 20:27:40(-0300)

All VUs finished. Total time: 10 seconds

--------------------------------
Summary report @ 20:27:45(-0300)
--------------------------------

http.codes.200: ..................................................... 1000
http.request_rate: .................................................. 82/sec
http.requests: ...................................................... 1000
http.response_time:
  min: ............................................................... 47
  max: ............................................................... 343
  median: ............................................................ 247.2
  p95: ............................................................... 333.7
  p99: ............................................................... 340.4
http.responses: ..................................................... 1000
vusers.completed: ................................................... 50
vusers.created: ..................................................... 50
vusers.created_by_name.0: ........................................... 50
vusers.failed: ...................................................... 0
vusers.session_length:
  min: ............................................................... 4599.7
  max: ............................................................... 5069.3
  median: ............................................................ 4965.3
  p95: ............................................................... 5065.6
  p99: ............................................................... 5065.6
```

Conclusiones:

La diferencia en el peso de la respuesta es de un 99,97%. Esto marca una mejora sustancial en la performance del servidor.

En los resultados del profiling con node --prof se ve que el proceso dura unos ticks más en el caso del proceso que utiliza console.log(), aunque se hace menos uso de librerías compartidas, lo que probablemente implique una mejora de rendimiento.

El profiling con test de carga con autocannon muestra que el proceso que utiliza console.log() tiene mayores latencias y sirve menos requests por segundo.

El profiling con node --inspect demuestra que los procesos bloqueantes ocupan la mayor parte del tiempo de trabajo del proceso.

El profiling con 0x demuestra, nuevamente, que los procesos que utilizan salidas por consola terminan siendo menos eficientes que aquellos que no lo hacen.

El test de carga con Artillery demuestra lo mismo que los tests de carga con autocannon, menor cantidad de requests servidas por segundo y mayor tiempo para hacerlo en el proceso que utiliza console.log().

Con todo esto se puede concluir que una buena compresión de las respuestas, junto con la optimización de los procesos, por ejemplo reduciendo console.log(), creando forks o clusters, y evitando ciclos innecesarios, permite que la velocidad de respuesta y la estabilidad del servidor escalen y soporten mayores cargas de uso.