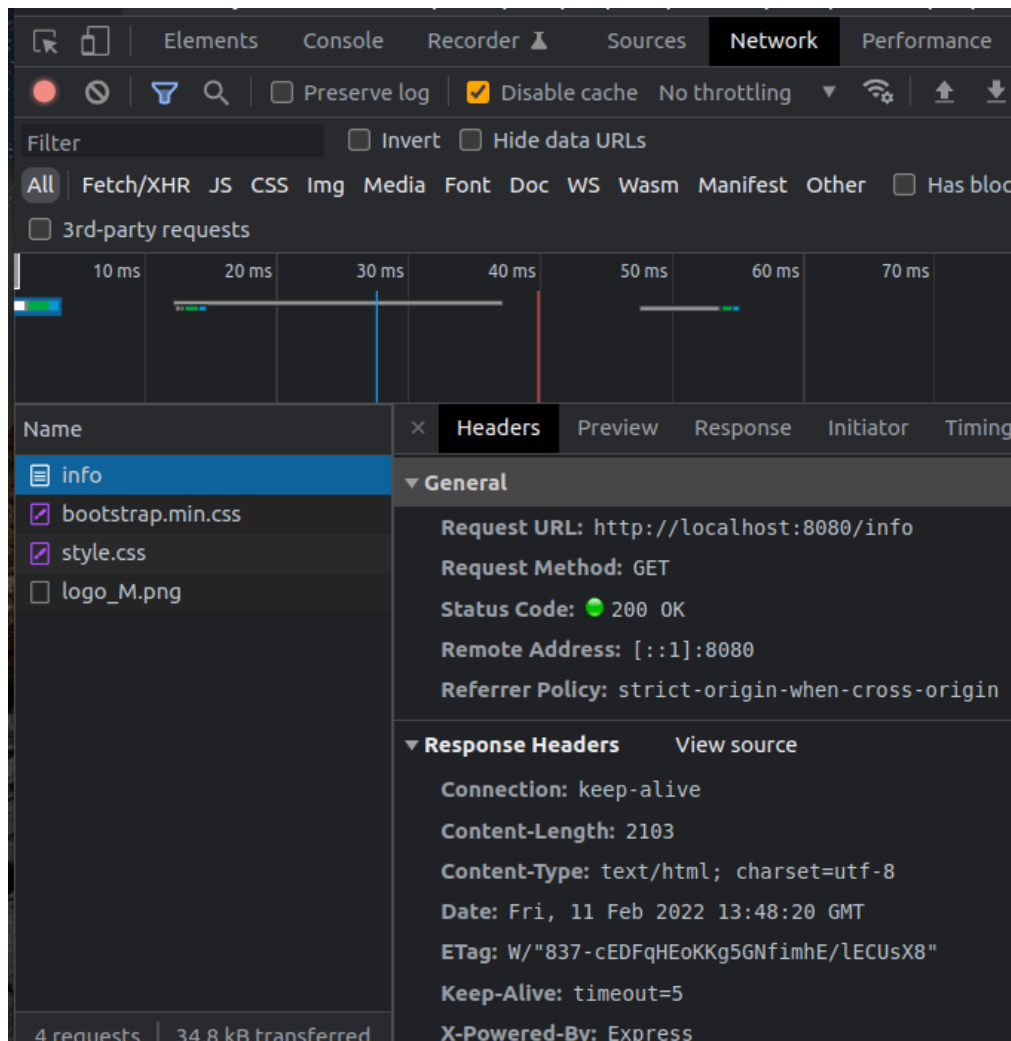


# Análisis de performance del servidor

## Compresión GZIP

Analizaremos sobre la ruta **/info** con y sin compresión, la diferencia de cantidad de bytes devueltos en un caso y otro.

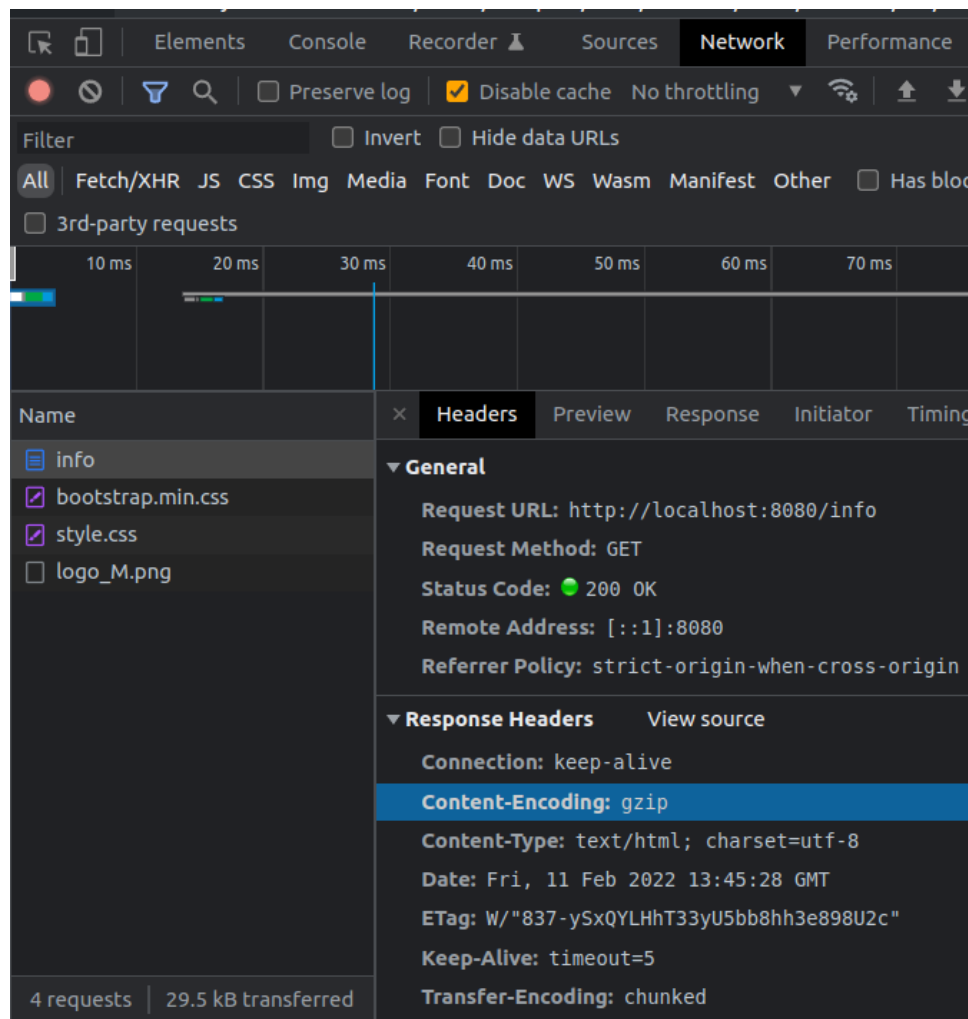
A continuación, se muestran los headers de la respuesta del servidor y el tamaño del paquete enviado cuando NO hay compresión.



El tamaño transferido del documento “info” es de **2.3KB** y el de “styles.css” de **6.5KB**.

Name	Status	Type	Initiator	Size	Time
info	200	document	Other	2.3 kB	2 ms
bootstrap.min.css	200	stylesheet	info	24.8 kB	66 ms
style.css	200	stylesheet	info	6.5 kB	2 ms
logo_M.png	200	png	Other	1.2 kB	2 ms

Ahora vemos los headers para el caso con compresión, activando el middleware **compression** para Express. Se puede ver el **Content-Encoding: gzip**.



En este caso, el tamaño transferido del documento “info” es de **1.3KB** y el de “styles.css” de **2.1KB**.

Name	Status	Type	Initiator	Size	Time
info	200	document	Other	1.3 kB	3 ms
bootstrap.min.css	200	stylesheet	info	24.8 kB	66 ms
style.css	200	stylesheet	info	2.1 kB	2 ms
logo_M.png	200	png	Other	1.2 kB	1 ms

El **porcentaje de compresión** en este caso es de un **43%** y un **68%** respectivamente, lo cual es mucho.

Hay que analizar cada caso que se nos platee, para ver si es más performante usar o no la compresión, porque si bien se logra disminuir el tamaño de la información transferida, esto es a costa de un mayor trabajo por parte del servidor (que debe realizar esta tarea). En casos de alto tráfico, no suele ser una buena opción.

## Profiling

Vamos a trabajar sobre la ruta `/info`, en modo **fork**, agregando o extrayendo un `console.log` de la información colectada antes de devolverla al cliente.

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

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

*a) Utilizamos como test de carga Artillery en línea de comandos, emulando 50 conexiones concurrentes con 20 request por cada una.*

Reporte de Artillery CON `console.log`:

```
Summary report @ 18:06:22(-0300)

http.codes.200: ..... 1000
http.request_rate: ..... 978/sec
http.requests: ..... 1000
http.response_time:
  min: ..... 0
  max: ..... 23
  median: ..... 5
  p95: ..... 8.9
  p99: ..... 13.1
http.responses: ..... 1000
vusers.completed: ..... 50
vusers.created: ..... 50
vusers.created_by_name.0: ..... 50
vusers.session_length:
  min: ..... 52
  max: ..... 183.1
  median: ..... 108.9
  p95: ..... 149.9
  p99: ..... 162.4
```

Reporte de Artillery SIN `console.log`:

```
Summary report @ 17:49:03(-0300)

http.codes.200: ..... 1000
http.request_rate: ..... 998/sec
http.requests: ..... 1000
http.response_time:
  min: ..... 0
  max: ..... 25
  median: ..... 3
  p95: ..... 6
  p99: ..... 8.9
http.responses: ..... 1000
vusers.completed: ..... 50
vusers.created: ..... 50
vusers.created_by_name.0: ..... 50
vusers.session_length:
  min: ..... 23.6
  max: ..... 123.2
  median: ..... 61
  p95: ..... 117.9
  p99: ..... 120.3
```

Se ve como la prueba SIN el console.log aumenta la performance, obteniéndose mejores tasas de respuestas/seg, tiempo medio de respuesta y duración de sesiones por usuario virtual.

El mismo comportamiento se ve también al analizar el resultado del profiling y teniendo en cuenta la cantidad de ticks que requirió una y otra prueba:

CON console.log

```
[Summary]:
```

ticks	total	nonlib	name
121	5.1%	26.2%	JavaScript
340	14.4%	73.6%	C++
236	10.0%	51.1%	GC
1895	80.4%		Shared libraries

SIN console.log

```
[Summary]:
```

ticks	total	nonlib	name
90	4.0%	21.3%	JavaScript
333	15.0%	78.7%	C++
191	8.6%	45.2%	GC
1803	81.0%		Shared libraries

b) Ahora utilizaremos Autocannon en línea de comandos, emulando 100 conexiones concurrentes realizadas en un tiempo de 20 segundos.

Reporte de Autocannon CON console.log:

```

npx autocannon -c 100 -d 20 "http://localhost:8080/info"
Running 20s test @ http://localhost:8080/info
100 connections

```

Stat	2.5%	50%	97.5%	99%	Avg	Stdev	Max
Latency	51 ms	58 ms	90 ms	110 ms	61.05 ms	11.23 ms	176 ms

Stat	1%	2.5%	50%	97.5%	Avg	Stdev	Min
Req/Sec	1009	1009	1671	1752	1622.75	167.09	1009
Bytes/Sec	2.38 MB	2.38 MB	3.94 MB	4.13 MB	3.83 MB	394 kB	2.38 MB

Req/Bytes counts sampled once per second.

33k requests in 20.03s, 76.5 MB read

Reporte de Autocannon SIN console.log:

```
npm run npx autocannon -c 100 -d 20 "http://localhost:8080/info"
Running 20s test @ http://localhost:8080/info
100 connections
```

Stat	2.5%	50%	97.5%	99%	Avg	Stdev	Max
Latency	39 ms	46 ms	99 ms	113 ms	52.34 ms	17.33 ms	168 ms

Stat	1%	2.5%	50%	97.5%	Avg	Stdev	Min
Req/Sec	1114	1114	1941	2047	1891	203.98	1114
Bytes/Sec	2.63 MB	2.63 MB	4.58 MB	4.83 MB	4.46 MB	481 kB	2.63 MB

Req/Bytes counts sampled once per second.

38k requests in 20.03s, 89.2 MB read

Nuevamente el mismo comportamiento mostrando mejor performance SIN el console.log. Se nota tanto en las peticiones totales dentro de los 20 segundos de test, en la latencia y en la tasa de respuestas por segundo.

El análisis del resultado del profiling también es concordante.

CON console.log

```
[Summary]:
```

ticks	total	nonlib	name
2243	11.0%	50.1%	JavaScript
2198	10.8%	49.1%	C++
1142	5.6%	25.5%	GC
15885	78.0%		Shared libraries

SIN console.log

```
[Summary]:
```

ticks	total	nonlib	name
2350	11.6%	51.6%	JavaScript
2182	10.8%	47.9%	C++
1070	5.3%	23.5%	GC
15681	77.5%		Shared libraries

## 2) El perfilamiento del servidor con el modo inspector de node.js --inspect.

a) Utilizamos como test de carga Artillery en línea de comandos, emulando 50 conexiones concurrentes con 20 request por cada una.

Reporte de Artillery CON console.log:

```
Summary report @ 22:01:08(-0300)

http.codes.200: ..... 1000
http.request_rate: ..... 646/sec
http.requests: ..... 1000
http.response_time:
  min: ..... 1
  max: ..... 103
  median: ..... 29.1
  p95: ..... 51.9
  p99: ..... 66
http.responses: ..... 1000
vusers.completed: ..... 50
vusers.created: ..... 50
vusers.created_by_name.0: ..... 50
vusers.session_length:
  min: ..... 154.3
  max: ..... 806.9
  median: ..... 671.9
  p95: ..... 772.9
  p99: ..... 788.5
```

Reporte de Artillery SIN console.log:

```
Summary report @ 22:04:40(-0300)

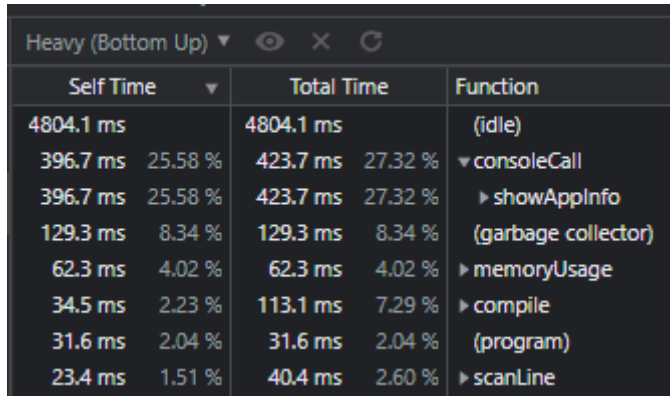
http.codes.200: ..... 1000
http.request_rate: ..... 970/sec
http.requests: ..... 1000
http.response_time:
  min: ..... 0
  max: ..... 15
  median: ..... 5
  p95: ..... 10.1
  p99: ..... 13.1
http.responses: ..... 1000
vusers.completed: ..... 50
vusers.created: ..... 50
vusers.created_by_name.0: ..... 50
vusers.session_length:
  min: ..... 60
  max: ..... 169.3
  median: ..... 117.9
  p95: ..... 156
  p99: ..... 162.4
```

Al igual que en el punto 1), se observa exactamente el mismo comportamiento de mayor performance en la variante SIN console.log en los mismos parámetros que citamos anteriormente.

El perfilamiento por **--inspect** es algo distinto al de **--perf** y **0x** (también usa **--perf** por dentro) y permite capturar mejor los stacks de las funciones. Pero también reduce la performance de la prueba en general. Se nota mucho más en el caso del console.log y también puede verificarse que se imprimen completamente todos los registros por consola en el navegador y de manera sincrónica.

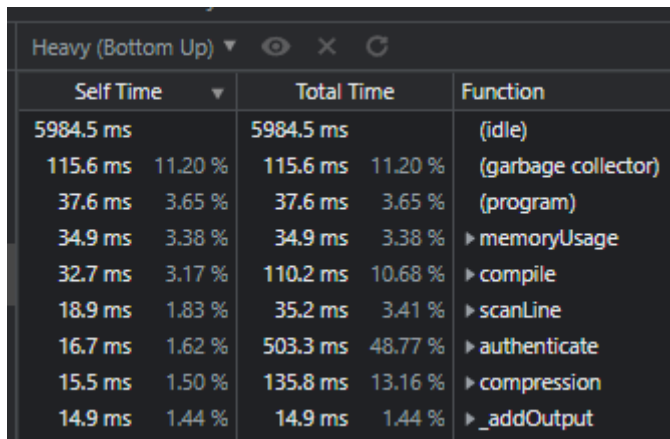
Analicemos los datos del stack que nos arroja inspect:

CON console.log:



Self Time	Total Time	Function
4804.1 ms	4804.1 ms	(idle)
396.7 ms 25.58 %	423.7 ms 27.32 %	▼ consoleCall
396.7 ms 25.58 %	423.7 ms 27.32 %	▶ showAppInfo
129.3 ms 8.34 %	129.3 ms 8.34 %	(garbage collector)
62.3 ms 4.02 %	62.3 ms 4.02 %	▶ memoryUsage
34.5 ms 2.23 %	113.1 ms 7.29 %	▶ compile
31.6 ms 2.04 %	31.6 ms 2.04 %	(program)
23.4 ms 1.51 %	40.4 ms 2.60 %	▶ scanLine

SIN console.log



Self Time	Total Time	Function
5984.5 ms	5984.5 ms	(idle)
115.6 ms 11.20 %	115.6 ms 11.20 %	(garbage collector)
37.6 ms 3.65 %	37.6 ms 3.65 %	(program)
34.9 ms 3.38 %	34.9 ms 3.38 %	▶ memoryUsage
32.7 ms 3.17 %	110.2 ms 10.68 %	▶ compile
18.9 ms 1.83 %	35.2 ms 3.41 %	▶ scanLine
16.7 ms 1.62 %	503.3 ms 48.77 %	▶ authenticate
15.5 ms 1.50 %	135.8 ms 13.16 %	▶ compression
14.9 ms 1.44 %	14.9 ms 1.44 %	▶ _addOutput

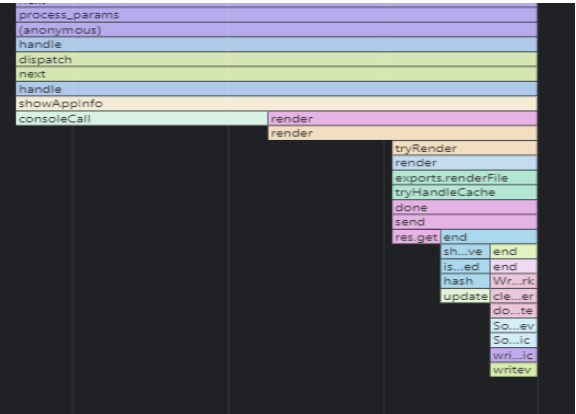
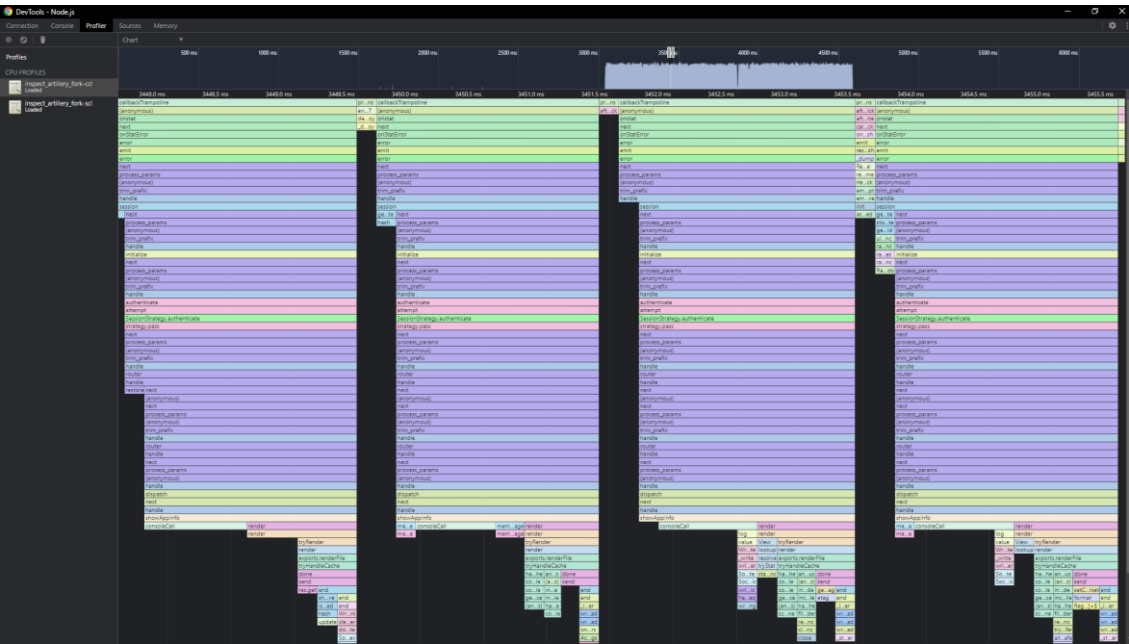
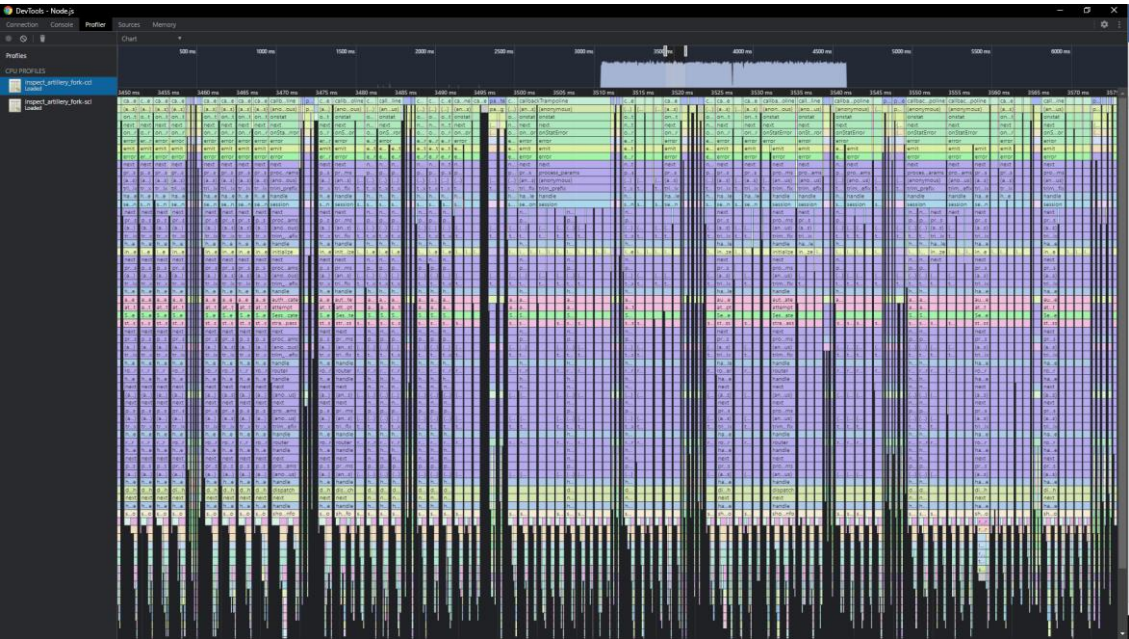
Se observa claramente como en el caso CON console.log, esta función es la de mayor tiempo propio y que permanece en la parte superior del stack, ya que se trata de una función síncrona bloqueante. Es llamada por el controlador **showAppInfo**.

Para el caso de SIN console.log, no se registran funciones de gran tiempo propio. Corresponde a un comportamiento asíncrono (no bloqueante).

Veamos las gráficas donde se pone en evidencia más claramente lo anterior.

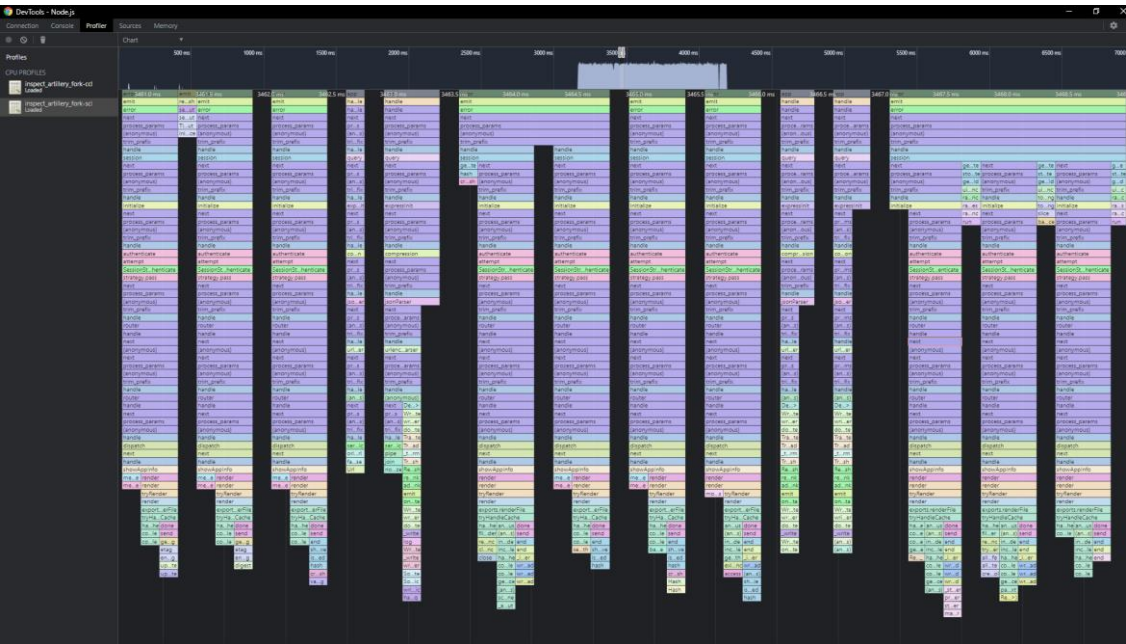


CON console.log





SIN console.log



```
next
process_params
(anonymous)
handle
dispatch
next
handle
showAppInfo
me...e render
me...e render
tryRender
render
export...erFile
tryHa...Cache
ha...he done
co...le send
co...le ge_g
etag
en_g
up...te
up...te
```

Ambos gráficos presentan la misma escala. Puede verse como el segundo (SIN console.log) es más “puntiagudo” indicando una naturaleza asíncrona. Mientras que el primero, posee picos más anchos resultando menos agudo. Debido a la naturaleza síncrona de la función console.log que durante su ejecución bloquea el stack de funciones.

Obviamente este bloqueo por los ms que dura la ejecución de la función, es lo que provoca que se ensanchen esos picos y cada petición demande más tiempo en ser completada. Disminuyendo la performance del servidor.

*b) Ahora utilizaremos Autocannon en línea de comandos, emulando 100 conexiones concurrentes realizadas en un tiempo de 20 segundos.*

Reporte de Autocannon CON console.log:

```

└─ npx autocannon -c 100 -d 20 "http://localhost:8080/info"
Running 20s test @ http://localhost:8080/info
100 connections

```

Stat	2.5%	50%	97.5%	99%	Avg	Stdev	Max
Latency	55 ms	104 ms	193 ms	218 ms	111.41 ms	45.87 ms	258 ms

Stat	1%	2.5%	50%	97.5%	Avg	Stdev	Min
Req/Sec	636	636	903	953	891.55	70.58	636
Bytes/Sec	1.5 MB	1.5 MB	2.13 MB	2.25 MB	2.1 MB	166 kB	1.5 MB

Req/Bytes counts sampled once per second.

18k requests in 20.02s, 42 MB read

Reporte de Autocannon SIN console.log:

```

└─ npx autocannon -c 100 -d 20 "http://localhost:8080/info"
Running 20s test @ http://localhost:8080/info
100 connections

```

Stat	2.5%	50%	97.5%	99%	Avg	Stdev	Max
Latency	50 ms	54 ms	90 ms	106 ms	57.9 ms	10.83 ms	170 ms

Stat	1%	2.5%	50%	97.5%	Avg	Stdev	Min
Req/Sec	1000	1000	1779	1889	1710.8	194.69	1000
Bytes/Sec	2.36 MB	2.36 MB	4.2 MB	4.46 MB	4.03 MB	459 kB	2.36 MB

Req/Bytes counts sampled once per second.

34k requests in 20.03s, 80.7 MB read

Otra vez evidencia el mismo comportamiento mostrando mejor performance SIN el console.log. Como antes, se nota tanto en las peticiones totales dentro de los 20 segundos de test, en la latencia y en la tasa de respuestas por segundo.

Si analizamos el resultado del inspect, tanto la tabla de las funciones como la gráfica muestran el mismo comportamiento.

Solo se presenta el resultado de las tablas, para no redundar en la misma información.

CON console.log:

Heavy (Bottom Up) ▾ 🔍 ✕ ↺			
Self Time ▾		Total Time	Function
8235.9 ms		8235.9 ms	(idle)
7224.5 ms	36.29 %	7606.5 ms	38.20 % ▶ consoleCall
1323.9 ms	6.65 %	1323.9 ms	6.65 % ▶ memoryUsage
666.4 ms	3.35 %	666.4 ms	3.35 % (garbage collector)
562.9 ms	2.83 %	1761.7 ms	8.85 % ▶ compile
334.6 ms	1.68 %	631.3 ms	3.17 % ▶ scanLine
330.7 ms	1.66 %	330.7 ms	1.66 % (program)
287.2 ms	1.44 %	15396.4 ms	77.33 % ▶ authenticate
263.3 ms	1.32 %	263.3 ms	1.32 % ▶ _addOutput

SIN console.log

Heavy (Bottom Up) ▾ 🔍 ✕ ↺			
Self Time ▾		Total Time	Function
4964.0 ms		4964.0 ms	(idle)
1197.0 ms	6.09 %	1197.0 ms	6.09 % ▶ memoryUsage
1037.2 ms	5.28 %	3170.6 ms	16.13 % ▶ compile
1019.6 ms	5.19 %	1019.6 ms	5.19 % (garbage collector)
607.5 ms	3.09 %	1122.0 ms	5.71 % ▶ scanLine
586.1 ms	2.98 %	586.1 ms	2.98 % (program)
554.8 ms	2.82 %	12062.5 ms	61.35 % ▶ authenticate
444.8 ms	2.26 %	444.8 ms	2.26 % ▶ _addOutput

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

Reporte de Autocannon CON console.log:

```
npx autocannon -c 100 -d 20 "http://localhost:8080/info"
Running 20s test @ http://localhost:8080/info
100 connections
```

Stat	2.5%	50%	97.5%	99%	Avg	Stdev	Max
Latency	51 ms	55 ms	95 ms	100 ms	59.5 ms	11.57 ms	175 ms

Stat	1%	2.5%	50%	97.5%	Avg	Stdev	Min
Req/Sec	947	947	1722	1893	1664.85	225.19	947
Bytes/Sec	2.23 MB	2.23 MB	4.06 MB	4.46 MB	3.93 MB	531 kB	2.23 MB

Req/Bytes counts sampled once per second.

33k requests in 20.03s, 78.5 MB read

Reporte de Autocannon SIN console.log:

```
npx autocannon -c 100 -d 20 "http://localhost:8080/info"
Running 20s test @ http://localhost:8080/info
100 connections
```

Stat	2.5%	50%	97.5%	99%	Avg	Stdev	Max
Latency	45 ms	47 ms	81 ms	93 ms	50.71 ms	9.94 ms	186 ms

Stat	1%	2.5%	50%	97.5%	Avg	Stdev	Min
Req/Sec	1106	1106	2051	2159	1951	254.58	1106
Bytes/Sec	2.61 MB	2.61 MB	4.84 MB	5.09 MB	4.6 MB	600 kB	2.61 MB

Req/Bytes counts sampled once per second.

39k requests in 20.03s, 92 MB read

El mismo comportamiento de siempre, mostrando mejor performance SIN el console.log. Se nota tanto en las peticiones totales dentro de los 20 segundos de test, en la latencia y en la tasa de respuestas por segundo.

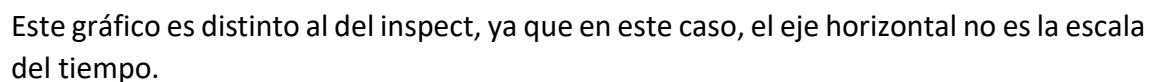
Nótese que los resultados son casi idénticos a cuando se utilizó **node --perf**. Y esto tiene que ver con que **0x** utiliza internamente este profiler como mencioné anteriormente cuando marcaba la diferencia con **node --inspect**.



```

/home/.nvm/versions/node/v16.18.0/bin/node /home/sereprec/Escritorio/CoderHouse/Backend/Clase32/defaio16-PrelezoSe
dispatch /home/sereprec/Escritorio/CoderHouse/Backend/Clase32/defaio16-PrelezoSe
handle /home/sereprec/Escritorio/CoderHouse/Backend/Clase32/defaio16-PrelezoSe
next /home/sereprec/Escritorio/CoderHouse/Backend/Clase32/defaio16-PrelezoSe
handle /home/sereprec/Escritorio/CoderHouse/Backend/Clase32/defaio16-PrelezoSe
log nc
/home/.nvm/versions/node/v16.18.0/bin/node View /hc
showAppInfo file:///home/sereprec/Escritorio/CoderHouse/Backend/Clase32/defaio16-PrelezoSe
dispatch /home/sereprec/Escritorio/CoderHouse/Backend/Clase32/defaio16-PrelezoSe
handle /home/sereprec/Escritorio/CoderHouse/Backend/Clase32/defaio16-PrelezoSe
next /home/sereprec/Escritorio/CoderHouse/Backend/Clase32/defaio16-PrelezoSe
handle /home/sereprec/Escritorio/CoderHouse/Backend/Clase32/defaio16-PrelezoSe

```

[illegible]

En el **gráfico de flama de 0x**, se representa una gráfica global de las funciones en el stack en base a los muestreos que se toman a lo largo de todo el test. No se grafican las instantáneas del stack a lo largo del tiempo, sino que se procesa esa información en conjunto y se trabaja en términos porcentuales. El ancho de los bloques corresponde al porcentaje de tiempo que dicha función aparece en esa posición del stack.

Los colores representan cuan “caliente” es una función en relación a su mayor permanencia en la parte superior del stack de funciones. Cuanto mayor sea el porcentaje de tiempo en esta parte, más caliente es.

Este comportamiento se asocia a las funciones síncronas bloqueantes, que permanecen mucho tiempo en la cima del stack bloqueando el proceso.

Puede verse como en el primer caso (CON console.log), se registra un color más “caliente” y se corresponde a la ejecución del console.log, cosa que no ocurre con el segundo de los casos donde las coloraciones, y por ende, la permanencia de las funciones en la cima del stack es más pareja (asincronismo).

**En resumen**, todos los test concuerdan en que la incorporación del console.log trae como resultado una baja apreciable de la performance del servidor.

Al analizar los números del profiling junto a los gráficos de flama de Ox y las gráficas del inspect, se llega a la conclusión de que lo anterior es producto que la función console.log es síncrona, bloqueando el proceso durante su ejecución y aumentando los tiempos de procesamiento de las solicitudes.