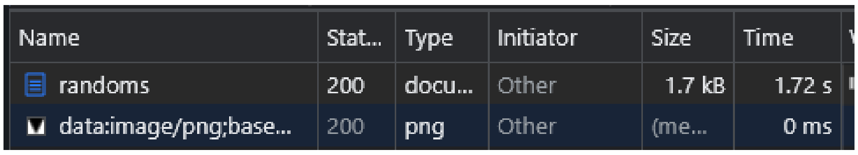
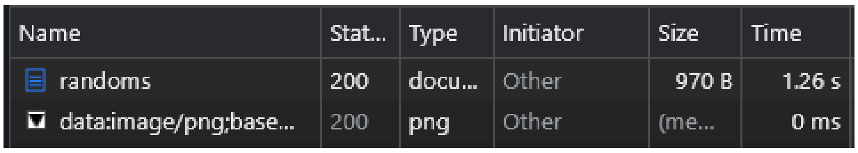
Compresión con gzip

Se utilizó “http://localhost:8080/api/randoms” para realizar la comparación.

Para la primera imagen no se aplicó compresión, se puede apreciar una diferencia de tamaño y tiempo de respuesta entre ambas solicitudes.



Name	Stat...	Type	Initiator	Size	Time
randoms	200	docu...	Other	1.7 kB	1.72 s
data:image/png;base...	200	png	Other	(me...	0 ms

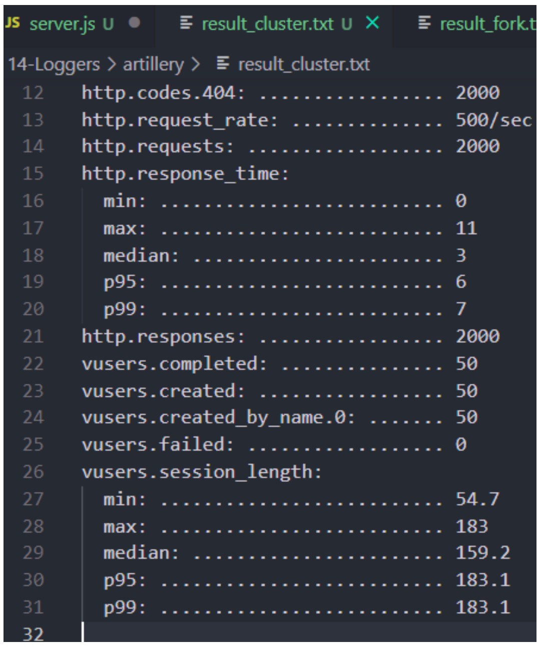


Name	Stat...	Type	Initiator	Size	Time
randoms	200	docu...	Other	970 B	1.26 s
data:image/png;base...	200	png	Other	(me...	0 ms

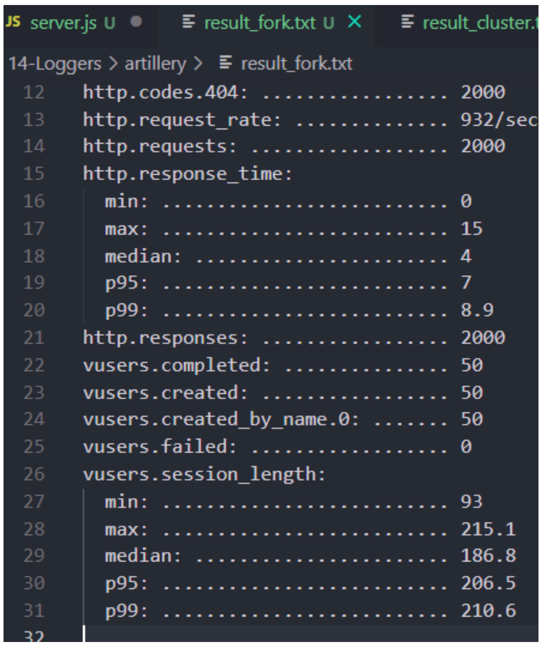
Al hacer uso de compresión, el tamaño se redujo en un 44% y el tiempo de respuesta en un 26%, demostrando que resulta bastante beneficioso el uso de este middleware.

Test de carga con Artillery

Se utilizó el comando “artillery quick --count 50 -n 40 http://localhost:8080/api/randoms?cant=9999999999999999999999999999 > 14-Loggers/artillery/filename.txt” para generar los documentos, estos se pueden encontrar en la carpeta del proyecto.



```
http.codes.404: ................. 2000
http.request_rate: .............. 500/sec
http.requests: .................. 2000
http.response_time:
  min: .......................... 0
  max: .......................... 11
  median: ....................... 3
  p95: .......................... 6
  p99: .......................... 7
http.responses: ................. 2000
vusers.completed: ............... 50
vusers.created: ................. 50
vusers.created_by_name.0: ....... 50
vusers.failed: .................. 0
vusers.session_length:
  min: .......................... 54.7
  max: .......................... 183
  median: ....................... 159.2
  p95: .......................... 183.1
  p99: .......................... 183.1
```



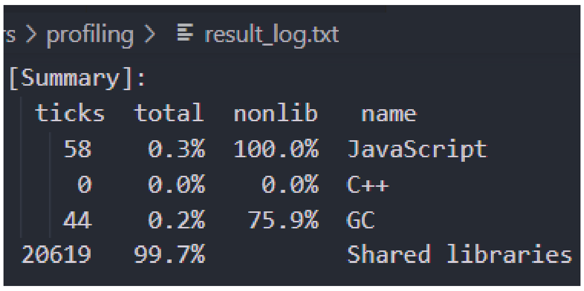
```
http.codes.404: ................. 2000
http.request_rate: .............. 932/sec
http.requests: .................. 2000
http.response_time:
  min: .......................... 0
  max: .......................... 15
  median: ....................... 4
  p95: .......................... 7
  p99: .......................... 8.9
http.responses: ................. 2000
vusers.completed: ............... 50
vusers.created: ................. 50
vusers.created_by_name.0: ....... 50
vusers.failed: .................. 0
vusers.session_length:
  min: .......................... 93
  max: .......................... 215.1
  median: ....................... 186.8
  p95: .......................... 206.5
  p99: .......................... 210.6
```

A la izquierda tenemos los resultados del servidor en modo Cluster, y a la derecha en modo Fork. Los tiempos de respuesta del primero son en general más bajos que los del segundo, por lo que es posible concluir que el modo Cluster es el más eficiente de ambos.

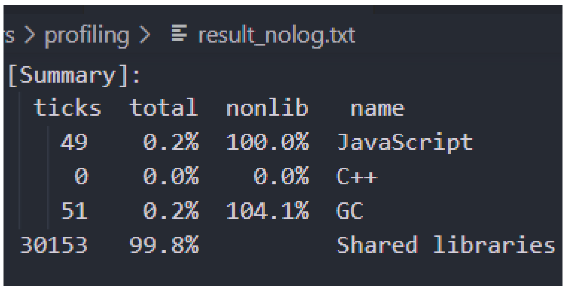
Perfilamiento del servidor

Las siguientes pruebas se realizan sobre “localhost:8080/api/info”, en modo fork, agregando o extrayendo un console.log de la información recolectada antes de devolverla al cliente.

1. --prof


s > profiling > result_log.txt

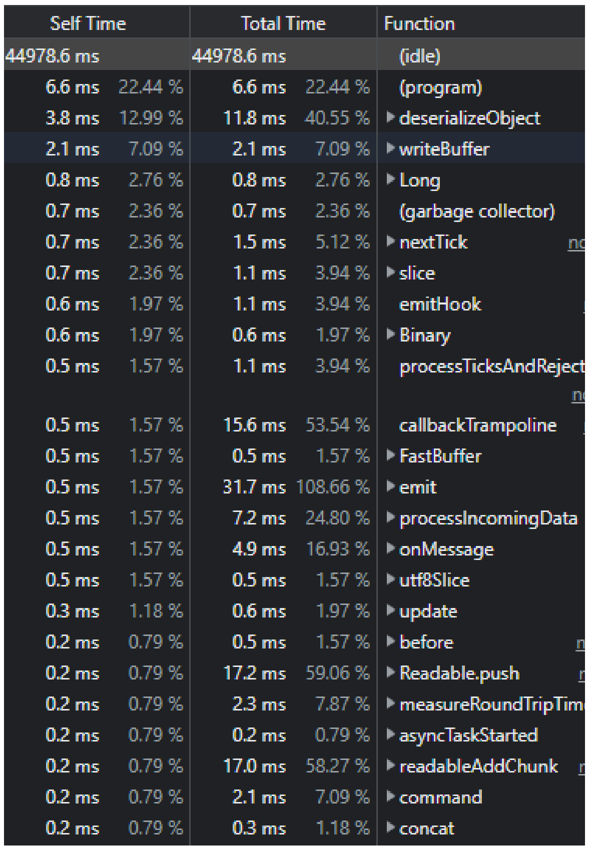
```
[Summary]:
 ticks  total  nonlib   name
   58    0.3%  100.0%  JavaScript
    0    0.0%    0.0%  C++
   44    0.2%   75.9%  GC
20619   99.7%          Shared libraries
```


s > profiling > result_nolog.txt

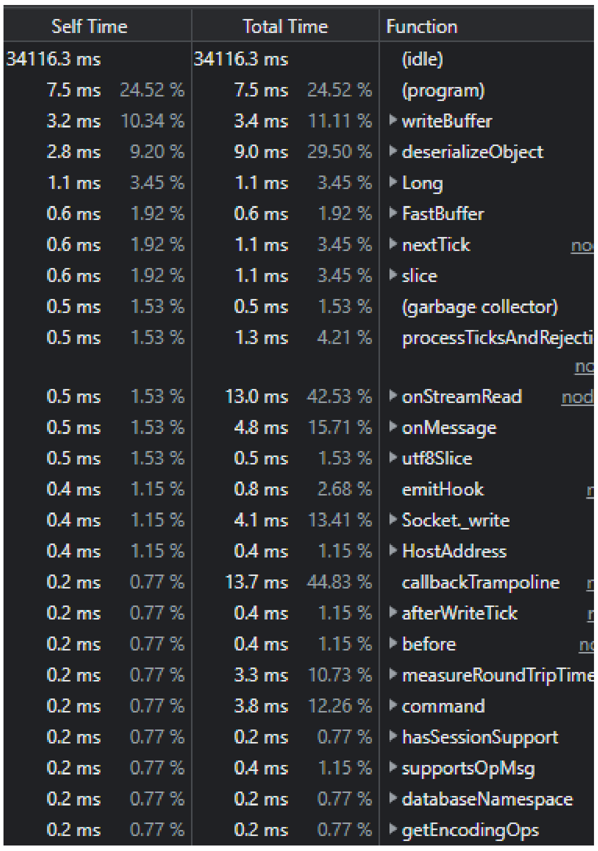
```
[Summary]:
 ticks  total  nonlib   name
   49    0.2%  100.0%  JavaScript
    0    0.0%    0.0%  C++
   51    0.2%  104.1%  GC
30153   99.8%          Shared libraries
```

En el resumen, vemos que el proceso que hace uso de un console.log tiene menos ticks. Ambos archivos pueden ser encontrados en la carpeta del proyecto.

2. --inspect



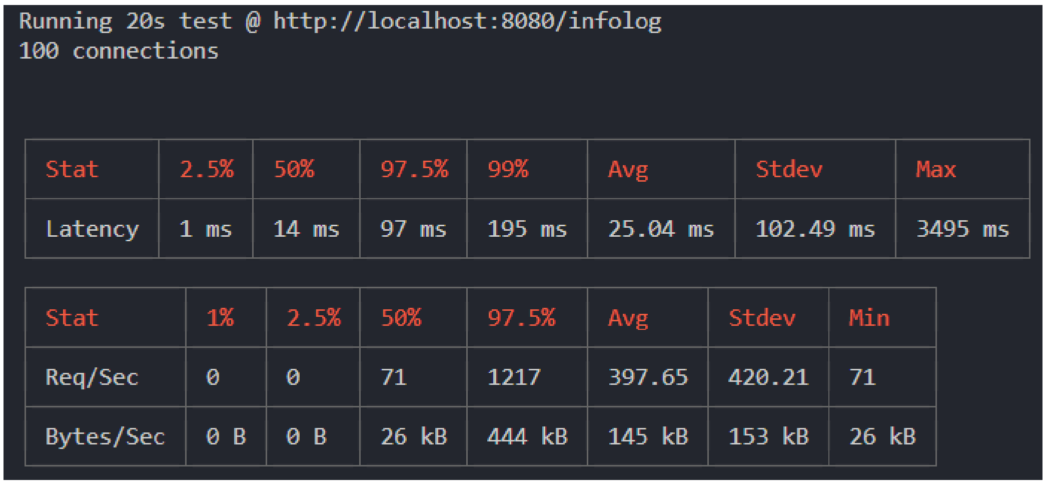
Self Time		Total Time		Function
44978.6 ms		44978.6 ms		(idle)
6.6 ms	22.44 %	6.6 ms	22.44 %	(program)
3.8 ms	12.99 %	11.8 ms	40.55 %	▸ deserializeObject
2.1 ms	7.09 %	2.1 ms	7.09 %	▸ writeBuffer
0.8 ms	2.76 %	0.8 ms	2.76 %	▸ Long
0.7 ms	2.36 %	0.7 ms	2.36 %	(garbage collector)
0.7 ms	2.36 %	1.5 ms	5.12 %	▸ nextTick
0.7 ms	2.36 %	1.1 ms	3.94 %	▸ slice
0.6 ms	1.97 %	1.1 ms	3.94 %	emitHook
0.6 ms	1.97 %	0.6 ms	1.97 %	▸ Binary
0.5 ms	1.57 %	1.1 ms	3.94 %	processTicksAndReject
0.5 ms	1.57 %	15.6 ms	53.54 %	callbackTrampoline
0.5 ms	1.57 %	0.5 ms	1.57 %	▸ FastBuffer
0.5 ms	1.57 %	31.7 ms	108.66 %	▸ emit
0.5 ms	1.57 %	7.2 ms	24.80 %	▸ processIncomingData
0.5 ms	1.57 %	4.9 ms	16.93 %	▸ onMessage
0.5 ms	1.57 %	0.5 ms	1.57 %	▸ utf8Slice
0.3 ms	1.18 %	0.6 ms	1.97 %	▸ update
0.2 ms	0.79 %	0.5 ms	1.57 %	▸ before
0.2 ms	0.79 %	17.2 ms	59.06 %	▸ Readable.push
0.2 ms	0.79 %	2.3 ms	7.87 %	▸ measureRoundTripTim
0.2 ms	0.79 %	0.2 ms	0.79 %	▸ asyncTaskStarted
0.2 ms	0.79 %	17.0 ms	58.27 %	▸ readableAddChunk
0.2 ms	0.79 %	2.1 ms	7.09 %	▸ command
0.2 ms	0.79 %	0.3 ms	1.18 %	▸ concat



Self Time		Total Time		Function
34116.3 ms		34116.3 ms		(idle)
7.5 ms	24.52 %	7.5 ms	24.52 %	(program)
3.2 ms	10.34 %	3.4 ms	11.11 %	▸ writeBuffer
2.8 ms	9.20 %	9.0 ms	29.50 %	▸ deserializeObject
1.1 ms	3.45 %	1.1 ms	3.45 %	▸ Long
0.6 ms	1.92 %	0.6 ms	1.92 %	▸ FastBuffer
0.6 ms	1.92 %	1.1 ms	3.45 %	▸ nextTick
0.6 ms	1.92 %	1.1 ms	3.45 %	▸ slice
0.5 ms	1.53 %	0.5 ms	1.53 %	(garbage collector)
0.5 ms	1.53 %	1.3 ms	4.21 %	processTicksAndRejecti
0.5 ms	1.53 %	13.0 ms	42.53 %	▸ onStreamRead
0.5 ms	1.53 %	4.8 ms	15.71 %	▸ onMessage
0.5 ms	1.53 %	0.5 ms	1.53 %	▸ utf8Slice
0.4 ms	1.15 %	0.8 ms	2.68 %	emitHook
0.4 ms	1.15 %	4.1 ms	13.41 %	▸ Socket._write
0.4 ms	1.15 %	0.4 ms	1.15 %	▸ HostAddress
0.2 ms	0.77 %	13.7 ms	44.83 %	callbackTrampoline
0.2 ms	0.77 %	0.4 ms	1.15 %	▸ afterWriteTick
0.2 ms	0.77 %	0.4 ms	1.15 %	▸ before
0.2 ms	0.77 %	3.3 ms	10.73 %	▸ measureRoundTripTime
0.2 ms	0.77 %	3.8 ms	12.26 %	▸ command
0.2 ms	0.77 %	0.2 ms	0.77 %	▸ hasSessionSupport
0.2 ms	0.77 %	0.4 ms	1.15 %	▸ supportsOpMsg
0.2 ms	0.77 %	0.2 ms	0.77 %	▸ databaseNamespace
0.2 ms	0.77 %	0.2 ms	0.77 %	▸ getEncodingOps

Nuevamente, se puede apreciar que el proceso que hace uso de un console.log (izquierda), toma más tiempo para completarse.

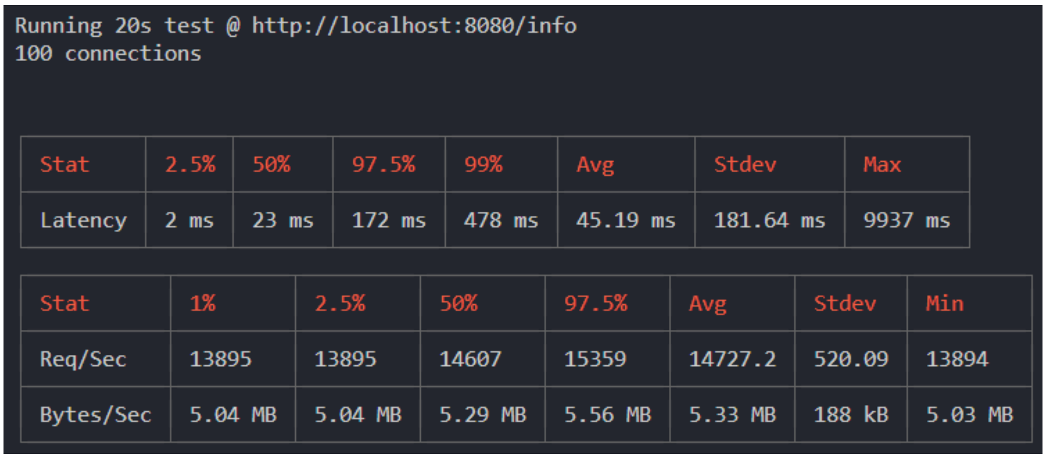
3. 0x



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

│ Stat    │ 2.5% │ 50%   │ 97.5% │ 99%    │ Avg      │ Stdev     │ Max     │
│ Latency │ 1 ms │ 14 ms │ 97 ms │ 195 ms │ 25.04 ms │ 102.49 ms │ 3495 ms │

│ Stat      │ 1%  │ 2.5% │ 50%   │ 97.5%  │ Avg    │ Stdev  │ Min   │
│ Req/Sec   │ 0   │ 0    │ 71    │ 1217   │ 397.65 │ 420.21 │ 71    │
│ Bytes/Sec │ 0 B │ 0 B  │ 26 kB │ 444 kB │ 145 kB │ 153 kB │ 26 kB │
```



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

│ Stat    │ 2.5% │ 50%   │ 97.5%  │ 99%    │ Avg      │ Stdev     │ Max     │
│ Latency │ 2 ms │ 23 ms │ 172 ms │ 478 ms │ 45.19 ms │ 181.64 ms │ 9937 ms │

│ Stat      │ 1%      │ 2.5%    │ 50%     │ 97.5%   │ Avg     │ Stdev  │ Min     │
│ Req/Sec   │ 13895   │ 13895   │ 14607   │ 15359   │ 14727.2 │ 520.09 │ 13894   │
│ Bytes/Sec │ 5.04 MB │ 5.04 MB │ 5.29 MB │ 5.56 MB │ 5.33 MB │ 188 kB │ 5.03 MB │
```

C:\WINDOWS\SYSTEM32\ntdll.dll [SHARED_LIB]
all stacks