Autor: Victor Vega
Curso: PROGRMACION BACKEND
Comision: 31005
Desafío 16: Logs, Profiling & Debug
Repositorio: Repositorio GutHub

PROFILING DE SERVIDOR:

De acuerdo a las consignas establecidas en este desafío, se ejecutaron pruebas o tests sobre el endpoint "/info" contenido en el archivo "server.js" del proyecto; cuyo repo se puede visualizar en el siguiente enlace: link al repo.

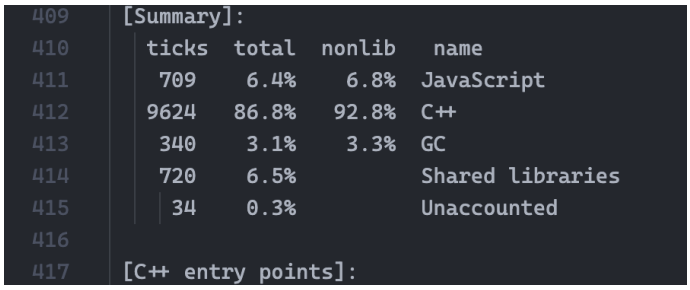
Con la finalidad de analizar los resultados del Profiling, en el endpoint se añadió un "console log" en una parte del código y se duplicó el código sin el respectivo "console log" para verificar el comportamiento en ambos casos del servidor.

La secuencia de testeo fue la siguiente:

1) Perfilamiento del servidor con—**prof de node.js**; en este caso se procedió a:
    a. Utilizar Artillery en la terminal, simulando 50 conexiones concurrentes con 20 request por cada conexión.
    b. Usar Autocannon en CLI, simulando 100 conexiones conjuntas realizadas en un tiempo de 20 segundos.
2) Perfilamiento del servidor con **–inspect de node.js**, usando el DevTools de Chrome para el análisis de los resultados.
3) Diagrama de flama con 0x, simulando la carga con Autocannon con los mismos parámetros anteriores.
4) Compresión con GZIP.

Profiling con '—prof' de node.js

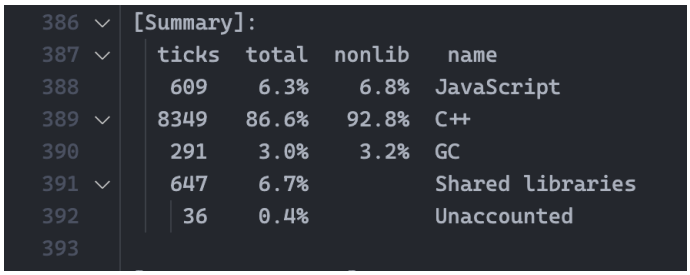
- Con Console Log:



```
[Summary]:
  ticks  total  nonlib   name
   709    6.4%    6.8%  JavaScript
  9624   86.8%   92.8%  C++
   340    3.1%    3.3%  GC
   720    6.5%          Shared libraries
    34    0.3%          Unaccounted

[C++ entry points]:
```

- Sin Console Log:



```
[Summary]:
  ticks  total  nonlib   name
   609    6.3%    6.8%  JavaScript
  8349   86.6%   92.8%  C++
   291    3.0%    3.2%  GC
   647    6.7%          Shared libraries
    36    0.4%          Unaccounted
```

Se puede observar que el perfilamiento un ahorro del 15% de ticks con el servidor sin "console log" a comparación con la carga con el "console log" en los eventos relativos con JavaScript.

Profiling '—prof' de node.js usando Artillery

- Con Console Log:

```
http.codes.200: ..... 103
http.codes.302: ..... 127
http.request_rate: ..... 61/sec
http.requests: ..... 274
http.response_time:
  min: ..... 2
  max: ..... 1458
  median: ..... 354.3
  p95: ..... 757.6
  p99: ..... 1043.3
http.responses: ..... 230
vusers.created: ..... 50
vusers.created_by_name.0: ..... 50
```

- Sin Console Log:

```
http.codes.200: ..... 9
http.codes.302: ..... 43
http.request_rate: ..... 87/sec
http.requests: ..... 87
http.response_time:
  min: ..... 0
  max: ..... 34
  median: ..... 4
  p95: ..... 18
  p99: ..... 22.9
http.responses: ..... 52
vusers.created: ..... 43
vusers.created_by_name.0: ..... 43
```

En los resultados obtenidos en la prueba con Artillery, se puede destacar que la carga sin "console log" fue significativamente menor en comparación a la carga con "console log"; esto visualizando el ratio de request y peticiones de conexión.

Profiling con '—prof' de node.js usando Autocannon

- Con Console Log:

```
● → Desafio16 git:(master) ✗ npx autocannon -c 100 -d 20 http://localhost:8080/info/test
Running 20s test @ http://localhost:8080/info/test
100 connections
```

Stat	2.5%	50%	97.5%	99%	Avg	Stdev	Max
Latency	685 ms	1019 ms	1846 ms	2038 ms	1052.03 ms	263.92 ms	2993 ms

Stat	1%	2.5%	50%	97.5%	Avg	Stdev	Min
Req/Sec	22	22	95	97	91.05	15.93	22
Bytes/Sec	8.66 kB	8.66 kB	37.4 kB	38.2 kB	35.8 kB	6.27 kB	8.66 kB

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

0 2xx responses, 1821 non 2xx responses
2k requests in 20.08s, 717 kB read
```

- Sin Console Log:

```
● → Desafio16 git:(master) ✗ 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	317 ms	1010 ms	1634 ms	1957 ms	984.96 ms	382.61 ms	3620 ms

Stat	1%	2.5%	50%	97.5%	Avg	Stdev	Min
Req/Sec	93	93	98	105	97.95	2.25	93
Bytes/Sec	36.6 kB	36.6 kB	38.6 kB	41.3 kB	38.5 kB	886 B	36.6 kB

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

0 2xx responses, 1959 non 2xx responses
2k requests in 20.09s, 771 kB read
```

Se observa que sin Console Log se procesan en promedio más requests por segundo que con Console Log incluido (97.95 vs 91.05 respectivamente). La latencia en promedio es menor en el servidor sin Console Log (984.96 vs 1052.03).

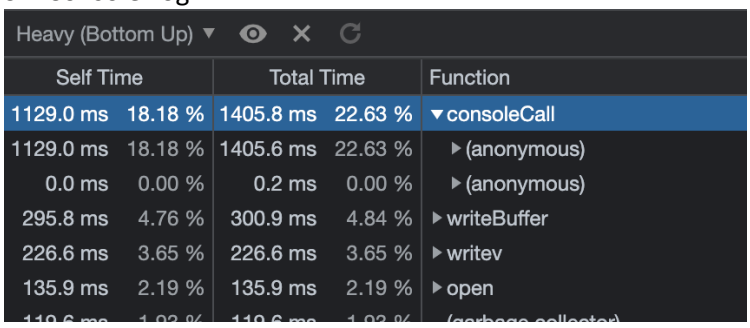
Perfilamiento del servidor con el modo --inspect de node.js.

Se puede visualizar en la imagen la información sobre el perfilamiento del servidor con el inspector de node (incluido Console Log en la ruta "/info") utilizando las DevTools de Chrome:

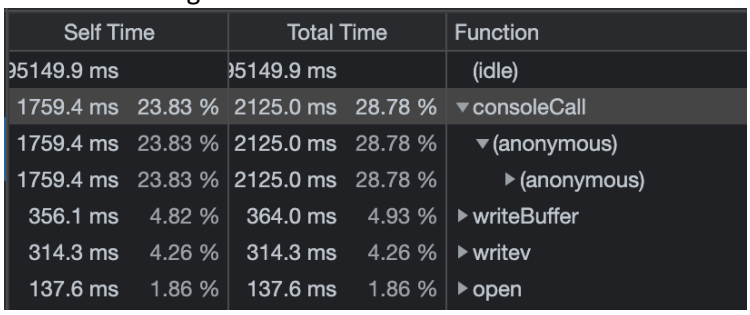
```
46   0.5 ms  infoRouter.get('', compression(), async (req, res) => {
47   1.1 ms      consoleLogger.info(req.baseUrl, req.method)
48
49   5.1 ms      if (req.user) {
50               const processInfo = [
51                   {name:"consoleArg", value: process.argv},
52                   {name: "platformName", value: process.platform},
53                   {name: "nodeVersion", value: process.version},
54                   {name: "memoryUsage", value: process.memoryUsage().rss},
55                   {name: "path", value: process.execPath},
56                   {name: "processId", value: process.pid},
57                   {name: "folder", value: process.cwd()},
58                   {name: "systemCores", value: numCPUs}
59               ];
60
61               return res.render('info', { processInfo });
62           };
63   6.5 ms      res.redirect('/login');
64       });
```

Es notorio que incluyendo el "Console Log" existe una pérdida de tiempo en el procesamiento de los requests realizdos en la prueba; el tiempo que tomó fue de 5.1 ms. Se evidencia este tiempo con la prueba hecha con Autocannon con los siguientes parámetros: una concurrencia de 100 requests durante 20 segundos.

- Sin Console Log:

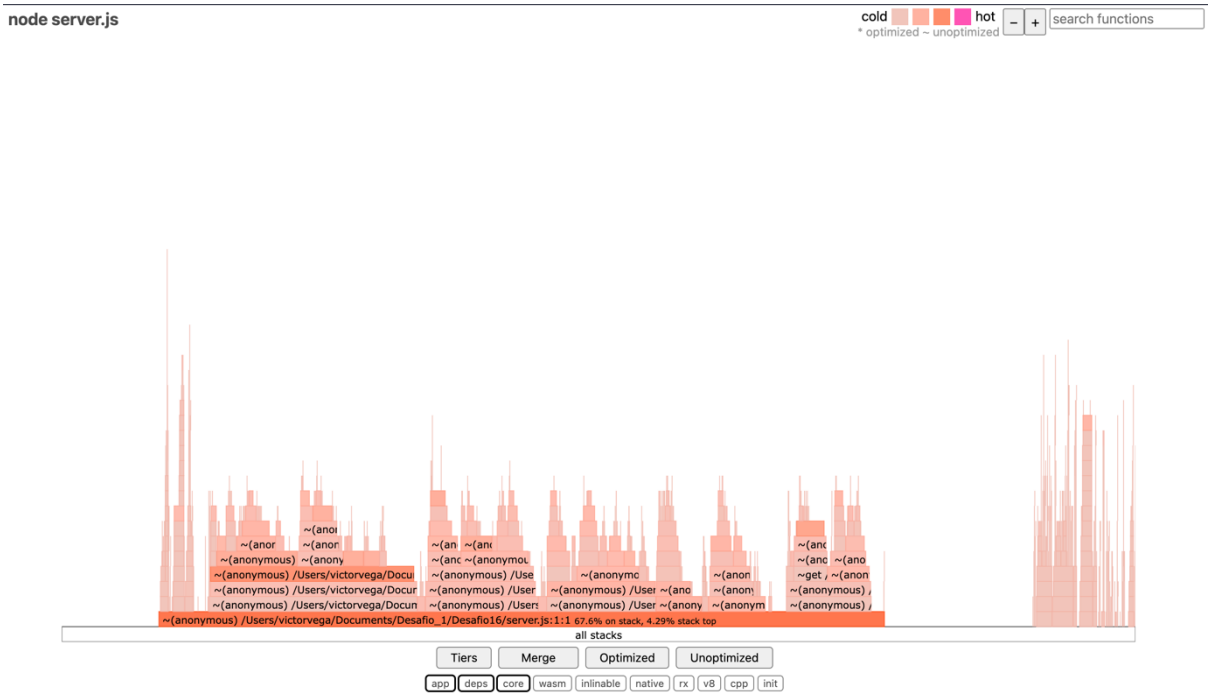


- Con Console Log:



Al analizar los resultados, el proceso que más tiempo toma es el de "consoleCall". Detalladamente este proceso incluye los "console logs" realizados por el servidor (realizados por log4js y los "console log" nativos propios del script). Se puede concluir de las imágenes, que el tiempo de procesamiento asociado a los "console logs" es mayor al añadir el "Console Log" en la ruta /info.

Diagrama de flama con 0x

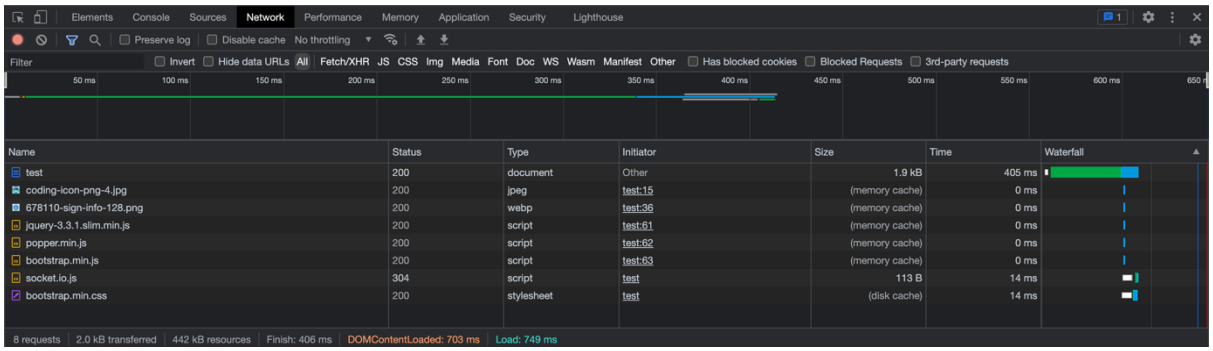


En la imagen se visualizan 2 picos de actividad relevantes. El primero de ellos relativo al test con “Autocannon” en la ruta “/info” sin “Console Log” y el segundo correspondiente al mismo test, hacia la misma ruta “/info” incluyendo “Console Log”.

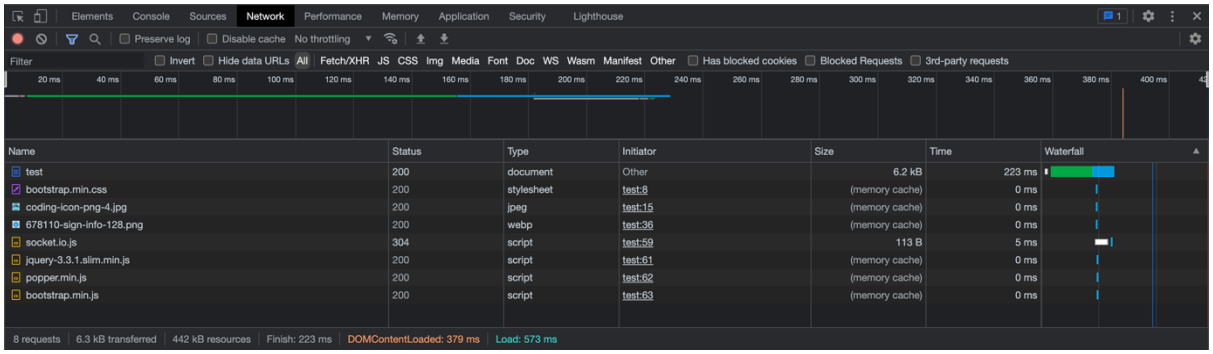
Es difícil de determinar en la imagen, sin embargo, lo que sí es notorio es la altura de la línea que incluye *“Console Log”* la cual es mayor al test en el que no se incluye dicho “Console Log”’. En ambos casos la consulta a “/info” y el “Console Log” (dependiendo del caso) se sitúan en lo más alto del gráfico de flama, en otras palabras, bloquean por un momento a los procesos de más abajo. Sin embargo, este bloqueo se produce por muy poco tiempo; por este motivo, no se nota un gran impacto en el gráfico (mesetas).

Compresión con GZip

- Comprimido:



- Sin Comprimir:



Se detalla en las imágenes anteriores que al usar compression el request a la ruta “/info” se transmiten 1.9 KB de información. Por otro lado, al no aplicar compression se transmiten 6.2 KB de información. Esto representa un ahorro de 70% aproximadamente en el tráfico de la información.

En conclusión, según los testeos realizados en cuanto al volumen de información se refiere, es recomendable la implementación de “compression”.