

Contenido

Pruebas de desempeño del servidor.....	2
Implementación de GZIP a través de la librería “compression”.	2
Pruebas de servidor haciendo uso de “Artillery” y la banderas de node.	4
Bandera “prof”	5
Bandera “inspect”	6
Librería Autocannon.....	8
Diagrama de flama - Librería 0x	9

Tabla de ilustraciones

Ilustración 1. Respuesta síncrona para 90 números usando GZIP (653 B).....	2
Ilustración 2. Respuesta síncrona para 90 números sin compresión (673 B).	2
Ilustración 3. Respuesta síncrona para 500 números usando GZIP (1.8 kB).....	3
Ilustración 4. Respuesta síncrona para 500 números sin compresión (5.3 kB).	3
Ilustración 5. Respuesta ruta /info usando GZIP (1.4 kB).	3
Ilustración 6. Respuesta ruta /info sin compresión (3.2 kB).	4
Ilustración 7. Resumen de resultados de la primera parte (680/1000) de las pruebas usando "Artillery" para conteo con console.log.	4
Ilustración 8. Conteo sin comando bloqueante.	5
Ilustración 9. Resumen de resultados de las pruebas usando "Artillery" para conteo sin console.log.	5
Ilustración 10. Resultado prueba bloqueante con la bandera "prof".	6
Ilustración 11. Resultado prueba no bloqueante con la bandera "prof".	6
Ilustración 12. Iniciar la inspección de la aplicación a través de la herramienta para desarrolladores de Chrome.	7
Ilustración 13. Inspección para prueba bloqueante.	7
Ilustración 14. Inspección para prueba no bloqueante.	8
Ilustración 15. Resultado prueba bloqueante mediante librería autocannon.....	8
Ilustración 16. Resultado prueba no bloqueante mediante librería autocannon.....	9
Ilustración 17. Diagrama de flama con instrucción bloqueante (slow).	10
Ilustración 18. Diagrama de flama sin instrucción bloqueante (fast).	10

Pruebas de desempeño del servidor

A continuación se presentarán los resultados de desempeño obtenidos mediante distintos mecanismos de prueba, entre los que se encuentran las librerías “autocannon” y “Ox” de node, así como el uso de compresión GZIP y el envío masivo de petición mediante la librería “artillery”.

Implementación de GZIP a través de la librería “compression”.

- Se implementó un nuevo uri al endpoint `/randoms/?` para el conteo aleatorio síncrono (sin fork) denominado `/randoms/sync?`
- Las diferencias en el tamaño de la compresión se empiezan a ver desde un comienzo, pero empiezan a ser beneficiosas solo alrededor de los 90-100 números. Por debajo, la compresión envía incluso más bytes que aquella sin compresión.

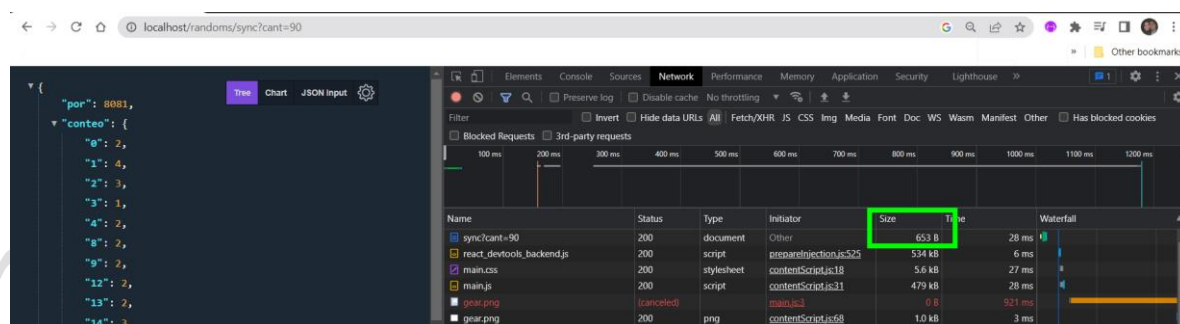


Ilustración 1. Respuesta síncrona para 90 números usando GZIP (653 B).

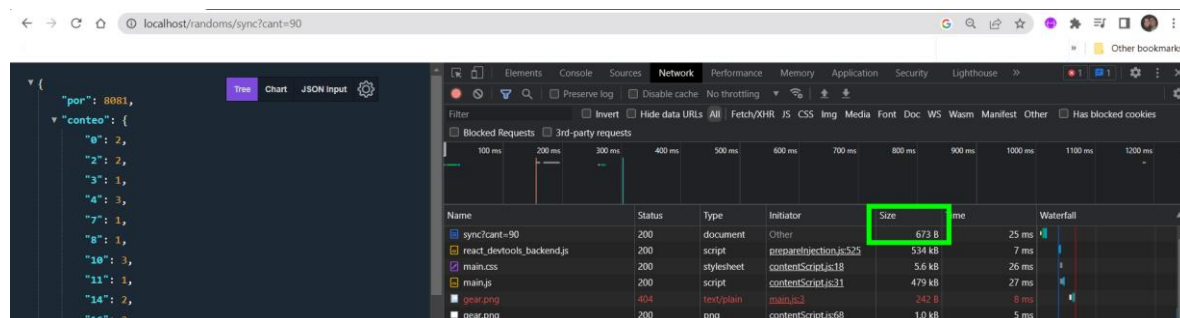


Ilustración 2. Respuesta síncrona para 90 números sin compresión (673 B).

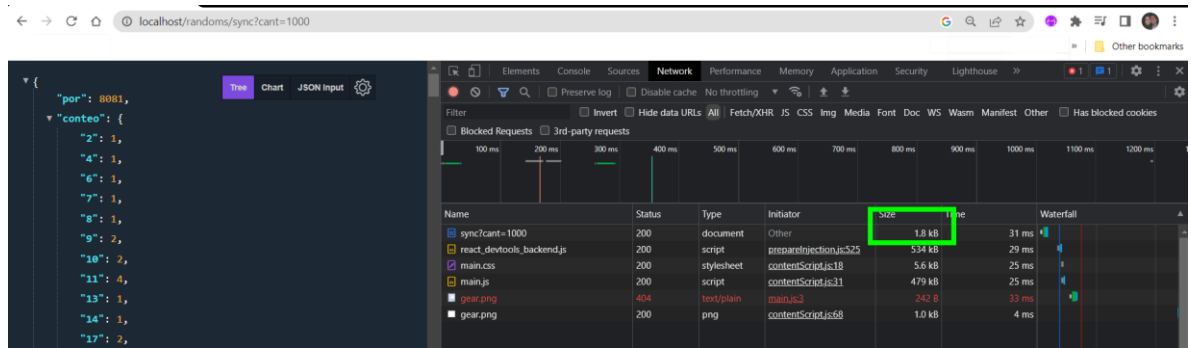


Ilustración 3. Respuesta síncrona para 500 números usando GZIP (1.8 kB).

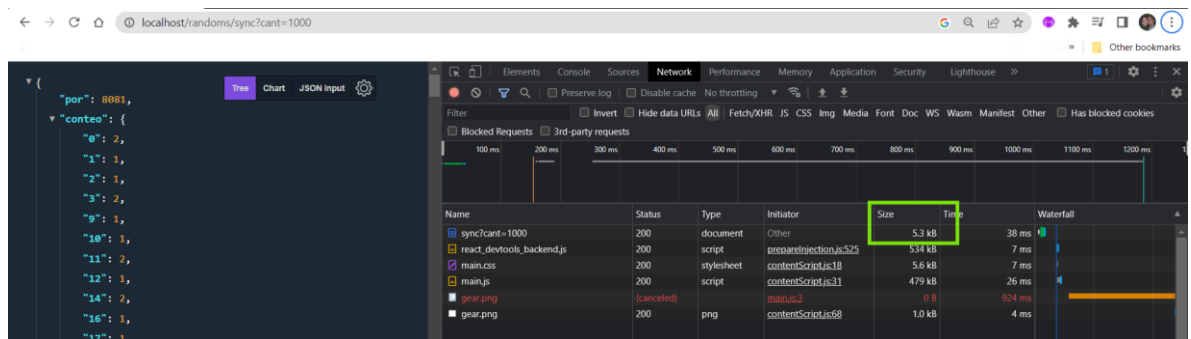


Ilustración 4. Respuesta síncrona para 500 números sin compresión (5.3 kB).

- Para la ruta `/info` la compresión es más evidente al ser una ruta que devuelve información cuyo tamaño es más estable:

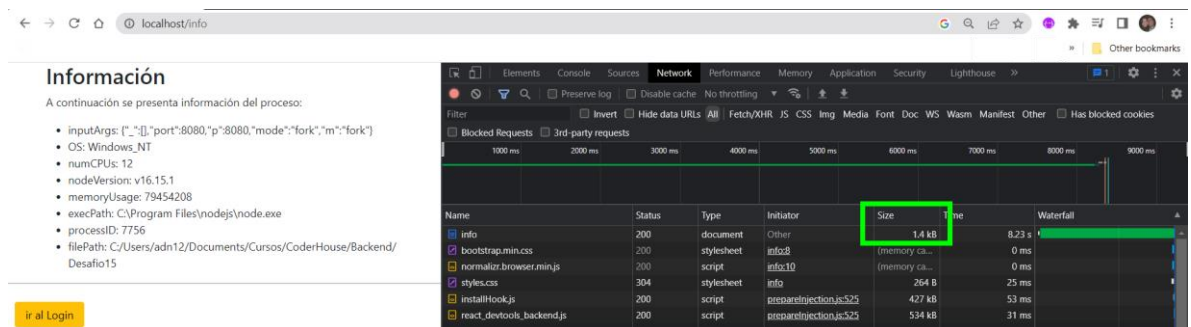


Ilustración 5. Respuesta ruta `/info` usando GZIP (1.4 kB).

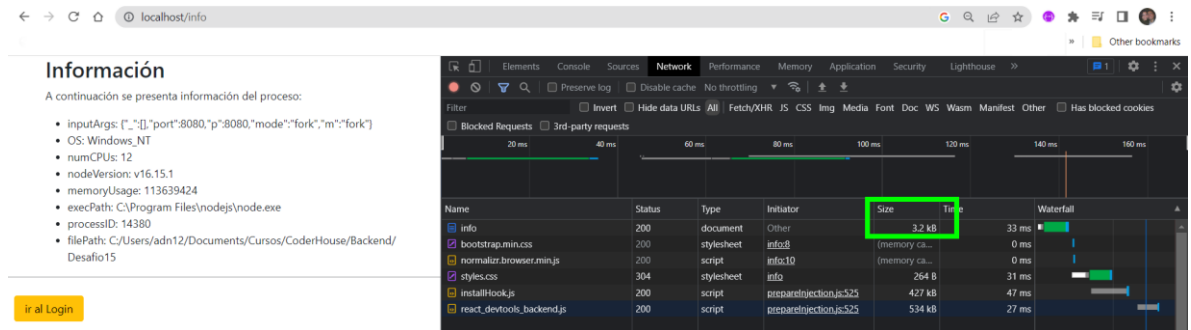


Ilustración 6. Respuesta ruta /info sin compresión (3.2 kB).

Pruebas de servidor haciendo uso de “Artillery” y la banderas de node.

Se instala la librería “artillery” con el comando `“npm -i -g artillery”`, luego se ejecuta el servidor con el comando `“node -prof server.js -port 8081”` y, posteriormente, en otra terminal, se realiza el lanzamiento de 100 peticiones (50 pruebas de 20 peticiones cada una) a la ruta de cálculo aleatorio (levantada anteriormente) con el comando `“artillery quick -c 50 -n 20 “http://localhost/randoms/sync?cant=20” > artillery_slow.txt”`. Los resultados se resumen a continuación:

```
Phase started: unnamed (index: 0, duration: 1s) 17:04:48(-0500)

Phase completed: unnamed (index: 0, duration: 1s) 17:04:49(-0500)

-----
Metrics for period to: 17:04:50(-0500) (width: 1.651s)
-----

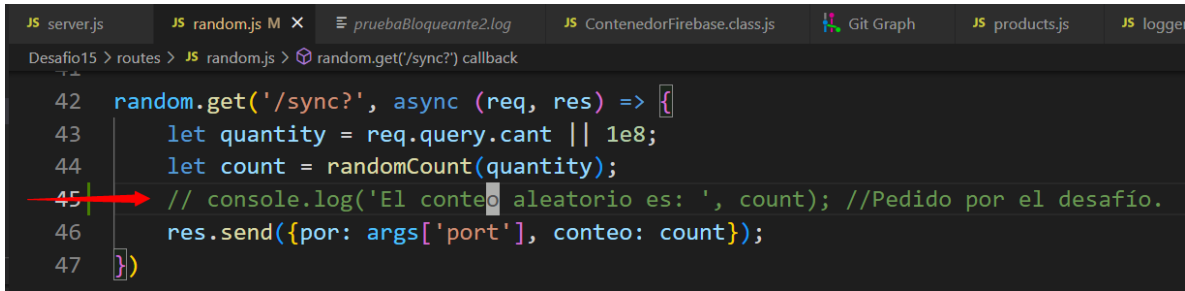
http.codes.200: ..... 680
http.request_rate: ..... 438/sec
http.requests: ..... 718
http.response_time:
  min: ..... 2
  max: ..... 137
  median: ..... 77.5
  p95: ..... 120.3
  p99: ..... 133
http.responses: ..... 680
```

Ilustración 7. Resumen de resultados de la primera parte (680/1000) de las pruebas usando “Artillery” para conteo con console.log.

Se observa que se tiene una tasa de respuesta de 438 peticiones por minuto, cada una a una mediana de 77.5 ms. Quiere decir que el 50% de las peticiones tomó menos tiempo y el otro 50% tomó mayor tiempo.

Para más información remitirse al archivo “artillery_slow.txt” del repositorio asociado.

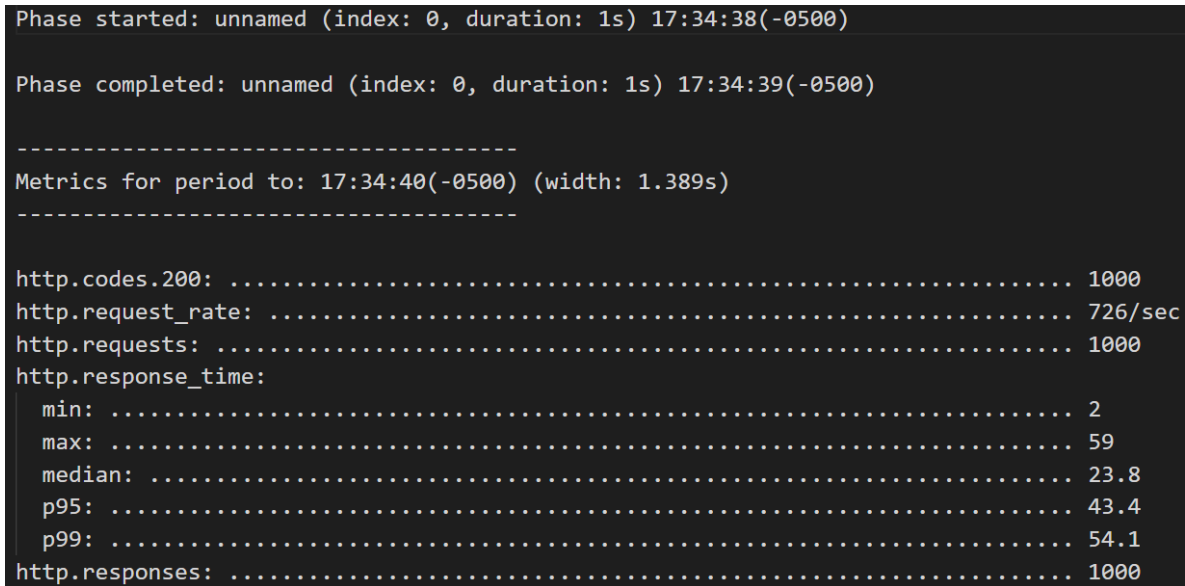
Si a la ruta de cálculo sincrónico se le quita el `console.log` implementado (función bloqueante), y se ejecuta la misma prueba (“`artillery quick -c 50 -n 20 "http://localhost/randoms/sync?cant=20" > artillery_fast.txt`”) se tiene el siguiente comportamiento:



```
42 random.get('/sync?', async (req, res) => {
43   let quantity = req.query.cant || 1e8;
44   let count = randomCount(quantity);
45   // console.log('El conteo aleatorio es: ', count); //Pedido por el desafío.
46   res.send({por: args['port'], conteo: count});
47 })
```

Ilustración 8. Conteo sin comando bloqueante.

Nota: Esta línea se debe comentar o añadir cada vez que se quiera probar la versión “bloqueante” (línea no comentada) y la versión “no bloqueante” (línea comentada).



```
Phase started: unnamed (index: 0, duration: 1s) 17:34:38(-0500)
Phase completed: unnamed (index: 0, duration: 1s) 17:34:39(-0500)

-----
Metrics for period to: 17:34:40(-0500) (width: 1.389s)
-----

http.codes.200: ..... 1000
http.request_rate: ..... 726/sec
http.requests: ..... 1000
http.response_time:
  min: ..... 2
  max: ..... 59
  median: ..... 23.8
  p95: ..... 43.4
  p99: ..... 54.1
http.responses: ..... 1000
```

Ilustración 9. Resumen de resultados de las pruebas usando “Artillery” para conteo sin `console.log`.

Se observa que se tiene una tasa de respuesta de 726 peticiones por minuto, cada una a una mediana de 23.8 ms, siendo el máximo tiempo de respuesta a una petición de 59 ms. Es decir, el máximo tiempo de respuesta de una petición fue menor (aproximadamente en un 20%) al tiempo promedio de las peticiones para la ruta con comando bloqueante (`console.log`). Esto con una sola instrucción de estas.

Para más información remitirse al archivo “`artillery_fast.txt`” del repositorio asociado.

Bandera “prof”

Para hacer uso de librerías nativas para analizar el desempeño del servidor, se debe pasar el archivo encriptado `.log` creado al ejecutar el servidor con la opción “`—prof`” a un archivo de texto. Esto se

hace mediante el comando “`node --prof-process pruebaBloqueante.log > prof_low.txt`”, y lo mismo para la prueba no bloqueante, con lo que obtenemos:

```
Statistical profiling result from pruebaBloqueante3.log, (2108 ticks, 0 unaccounted, 0
excluded).

[Shared libraries]:
  ticks  total  nonlib   name
    1684    79.9%         C:\WINDOWS\SYSTEM32\ntdll.dll
     413    19.6%         C:\Program Files\nodejs\node.exe
       1     0.0%         C:\WINDOWS\System32\KERNEL32.DLL

[JavaScript]:
  ticks  total  nonlib   name
     2     0.1%    20.0% LazyCompile: *trim_prefix
C:\Users\adn12\Documents\Cursos\CoderHouse\Backend\Desafio15\node_modules\express\li
b\router\index.js:293:23
     2     0.1%    20.0% LazyCompile: *resolve node:path:158:10
     2     0.1%    20.0% LazyCompile: *next
C:\Users\adn12\Documents\Cursos\CoderHouse\Backend\Desafio15\node_modules\express\li
b\router\index.js:177:16
```

Ilustración 10. Resultado prueba bloqueante con la bandera "prof".

```
Statistical profiling result from pruebasNoBloqueante.log, (25563 ticks, 0 unaccounted, 0
excluded).

[Shared libraries]:
  ticks  total  nonlib   name
  25258   98.8%         C:\WINDOWS\SYSTEM32\ntdll.dll
    286    1.1%         C:\Program Files\nodejs\node.exe
       5     0.0%         C:\WINDOWS\System32\KERNELBASE.dll
       2     0.0%         C:\WINDOWS\System32\KERNEL32.DLL

[JavaScript]:
  ticks  total  nonlib   name
     2     0.0%   16.7% Function: ^processTimers node:internal/timers:487:25
     1     0.0%    8.3% LazyCompile: *resolve node:path:158:10
     1     0.0%    8.3% LazyCompile: *pushAsyncContext node:internal/async_hooks:540:26
     1     0.0%    8.3% LazyCompile: *next
C:\Users\adn12\Documents\Cursos\CoderHouse\Backend\Desafio15\node_modules\express\li
b\router\index.js:177:16
```

Ilustración 11. Resultado prueba no bloqueante con la bandera "prof".

Observamos que la prueba bloqueante tiene mucho menos cantidad de “ticks” (alrededor del 7%) que aquella de la prueba no bloqueante. Esto quiere decir que el procesador procesa menos información, ya que los “ticks” son los tiempos del reloj del procesador.

Para más información remitirse a los archivos “`prof_fast.txt`” y “`prof_slow.txt`” del repositorio asociado.

Bandera “inspect”

Se puede realizar un análisis del desempeño del servidor haciendo uso de las librerías nativas también mediante la bandera “`—inspect`”. Se ejecuta el servidor con esta bandera y se utiliza los

complementos de desarrollador del navegador (Chrome:\\Inspect) para monitorear la ejecución de las tareas de la aplicación:

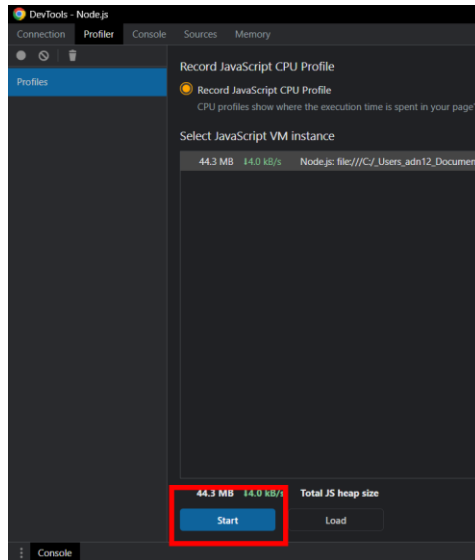


Ilustración 12. Iniciar la inspección de la aplicación a través de la herramienta para desarrolladores de Chrome.

Luego de iniciar la inspección, se ejecuta nuevamente el comando de artillery mostrado anteriormente:

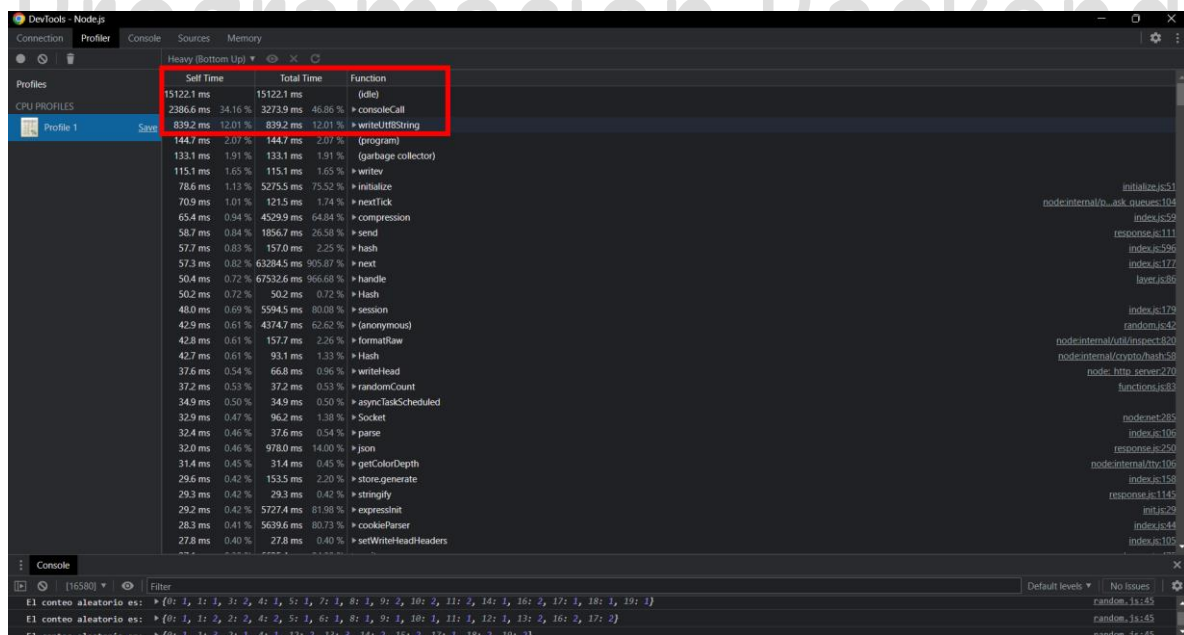


Ilustración 13. Inspección para prueba bloqueante.

Se observa que, cuando no está esperando (idle), la mayoría del tiempo la aplicación se encuentra escribiendo ya sea en consola o en pantalla (console.call-34%+ writeUt8String-12%) con el 36% del tiempo total.

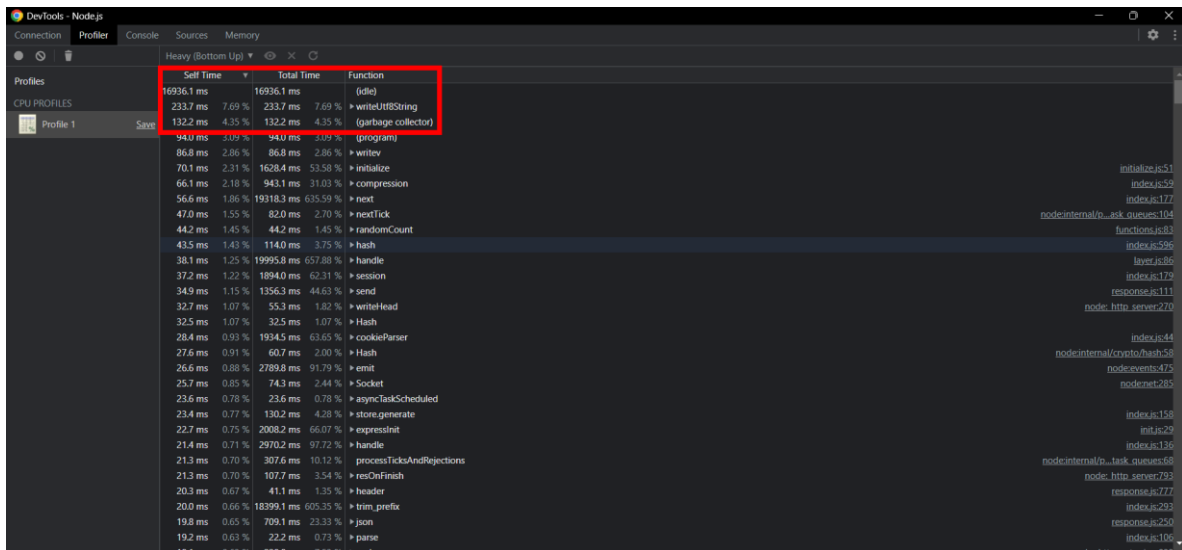


Ilustración 14. Inspección para prueba no bloqueante.

Para el caso no bloqueante, se observa que la mayoría del tiempo que la aplicación no está esperando (idle) lo utiliza para escribir en pantalla, pero solo el 7.7% sin invertir tiempo escribiendo en consola.

Librería Autocannon.

Se instala la librería “autocannon” para realizar una prueba de desempeño mediante la línea de comandos. Se utiliza el comando “`npm test`” luego de haber modificado el script “test” del “package.json” a “`test: node benchmark.js`”. Al ejecutar el comando tenemos:

```
> desafio5-ejs@1.0.0 test
> node benchmark.js

2023-02-15T01:01:06.491Z - info: Inicio de pruebas Benchmark para http://localhost/randoms/sync?cant=20
2023-02-15T01:01:26.670Z - info: Fin de pruebas Benchmark para http://localhost/randoms/sync?cant=20
Running 20s test @ http://localhost/randoms/sync?cant=20
100 connections
```

Stat	2.5%	50%	97.5%	99%	Avg	Stdev	Max
Latency	2046 ms	2061 ms	2075 ms	2079 ms	2060.92 ms	10.61 ms	2087 ms

Stat	1%	2.5%	50%	97.5%	Avg	Stdev	Min
Req/Sec	0	0	0	100	45	49.75	100
Bytes/Sec	0 B	0 B	0 B	67.6 kB	30.4 kB	33.6 kB	67.6 kB

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

0 2xx responses, 900 non 2xx responses
1k requests in 20.15s, 608 kB read
```

Ilustración 15. Resultado prueba bloqueante mediante librería autocannon

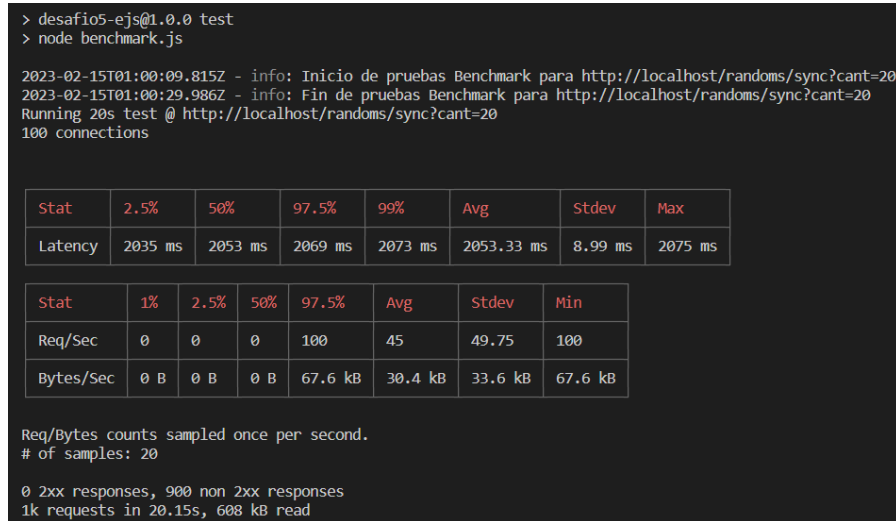


Ilustración 16. Resultado prueba no bloqueante mediante librería autocannon.

Confirmamos nuevamente que los tiempos de respuesta cuando no se tiene la instrucción bloqueante (console.log) es menor, en este caso por una media de 8 ms, con respecto a cuando se tiene la instrucción.

Diagrama de flama - Librería Ox

Se realiza el diagrama de flama para comparar el desempeño de la ruta “/randoms/sync?cant=20” del aplicativo siguiendo los siguientes pasos:

- Se instala la librería Ox y se ejecuta el servidor mediante el comando “`Ox server.js -port 8081`”.
- En otra terminal se ejecuta el comando “`artillery quick -c 100 -n 20 "http://localhost/randoms/sync?cant=20" > artillery_slow_Ox.txt`” para enviar las peticiones que formarán el diagrama de flama.
- Se finaliza el proceso de la primera terminal creandose la carpeta que contiene el diagrama de flama (posteriormente renombrada como “slow.Ox” o “fast.Ox” dependiendo del caso) que se muestra a continuación:

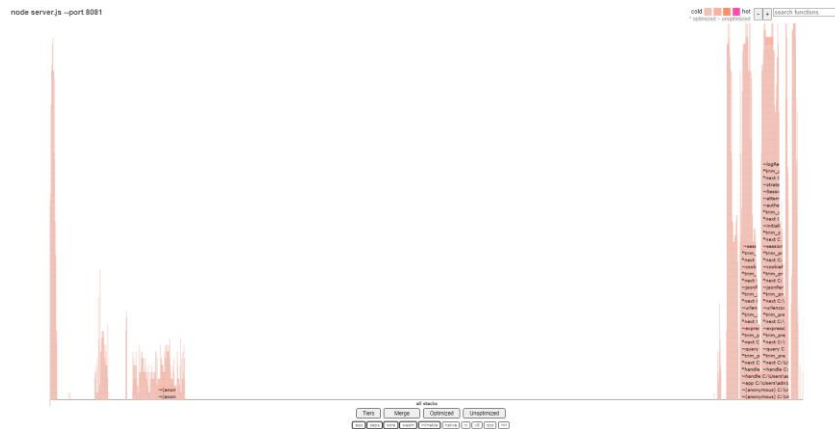


Ilustración 17. Diagrama de flama con instrucción bloqueante (slow).

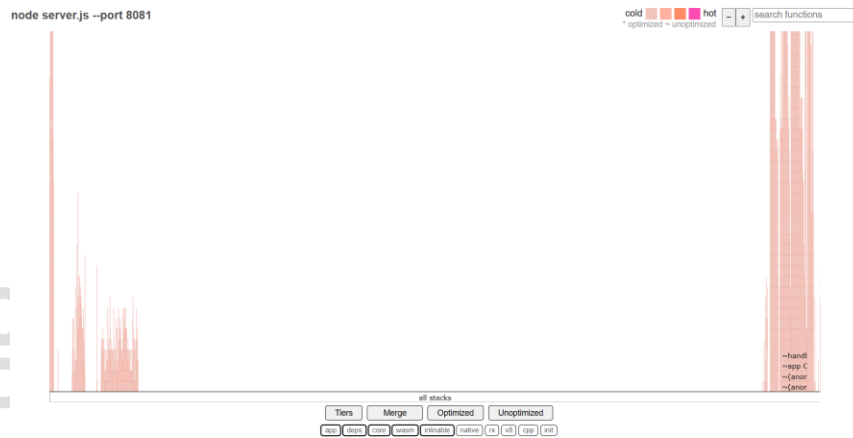


Ilustración 18. Diagrama de flama sin instrucción bloqueante (fast).

La parte izquierda del gráfico comprende el proceso de carga del servidor (carga de node y arranque). Esta etapa no presenta ningún cambio, sin embargo, la parte derecha muestra el comportamiento del servidor a la llegada de las peticiones enviadas por Artillery. Observamos en la primera gráfica (slow) que el ancho de dichas barras es mayor, implicando que los procesos duraron más tiempo. Esto se debe a la instrucción `console.log` implementada, la cual es bloqueante. Por su parte, a la derecha de la segunda gráfica se muestran barras mucho más delgadas, pero más altas, indicando así la persistencia de procesos más rápidos y menos bloqueantes.

Para más información remitirse a los archivos “flamegraph.html” de las carpetas “slow.0x” y “fast.0x”, respectivamente, del repositorio asociado.

Conclusiones

Podemos concluir que el uso de instrucciones síncronas impacta de negativamente y de forma muy clara el desempeño del servidor. Se debe evitar en gran medida su uso y para esto son útiles las instrucciones asíncronas basadas en promesas, así como las librerías asíncronas como “log4Js”,

“Winston” o “Pino”, que nos permiten hacer uso de mensajes que no impactan el rendimiento del servidor. De igual manera, las librerías de pruebas de desempeño como son “Ox”, “autocannon” o “artillery” son fundamentales en el proceso de identificación de cuellos de botella o demás problemas de desempeño en nuestro servidor.

Se debe tener en cuenta también el tamaño de la información transmitida para lo cual se debe considerar el uso de librerías de compresión como “compression”, no olvidando que es posible que su uso tenga resultados negativos en el rendimiento, razón por la cual se aconseja probar su implementación.

Programación Backend