

Informe de Resultados

Desafío 14: Loggers, Gzip y Analisis de Performance

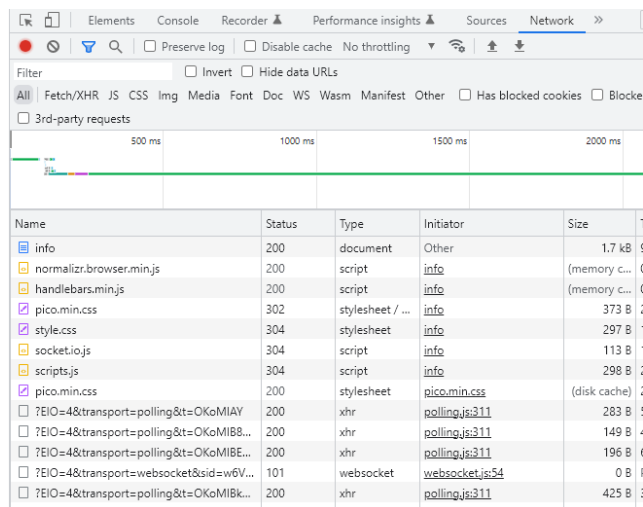
1. Verificar sobre la ruta /info con y sin compresión, la diferencia de cantidad de bytes devueltos en un caso y otro.

Sin compresión

La cantidad de bytes devueltos es de **1.7KB** como se puede observar en la Figura 1

Info del sistema

- Argumentos de Entrada:
- Path de ejecucion: C:\Program Files\nodejs\node.exe
- Nombre de la Plataforma (sistema operativo): win32
- Process ID: 9980
- Version de Node.JS: v16.14.0
- Carpeta del proyecto:
C:\Users\Gab_m\Documents\Cursos\Programacion-backend\Desafios\Desafio14-Analisis-De-Performance
- Memoria Total Reservada (rss): 101797888
- Cantidad de procesadores: 4



Name	Status	Type	Initiator	Size
info	200	document	Other	1.7 KB
normalizr.brower.min.js	200	script	info	(memory cache)
handlebars.min.js	200	script	info	(memory cache)
pico.min.css	302	stylesheet / ...	info	373 B
style.css	304	stylesheet	info	297 B
socket.io.js	304	script	info	113 B
scripts.js	304	script	info	298 B
pico.min.css	200	stylesheet	pico.min.css	(disk cache)
?EIO=4&transport=polling&tt=OKoMIAY	200	xhr	polling.js:311	283 B
?EIO=4&transport=polling&tt=OKoMIB8...	200	xhr	polling.js:311	149 B
?EIO=4&transport=polling&tt=OKoMIB8...	200	xhr	polling.js:311	196 B
?EIO=4&transport=websocket&sid=w6V...	101	websocket	websocket.js:54	0 B
?EIO=4&transport=polling&tt=OKoMIB8...	200	xhr	polling.js:311	425 B

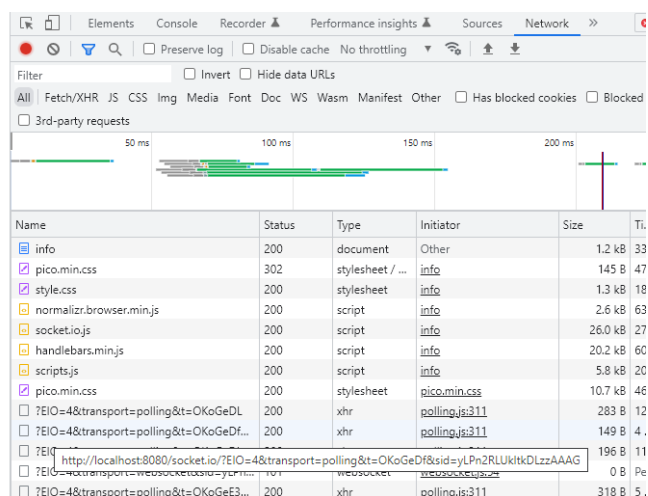
Figura 1. Captura de pantalla de cantidad de bytes sin compresión

Con compresión

La cantidad de bytes devueltos es de **1.2KB** como se puede observar en la Figura 2

Info del sistema

- Argumentos de Entrada:
- Path de ejecucion: C:\Program Files\nodejs\node.exe
- Nombre de la Plataforma (sistema operativo): win32
- Process ID: 3760
- Version de Node.JS: v16.14.0
- Carpeta del proyecto:
C:\Users\Gab_m\Documents\Cursos\Programacion-backend\Desafios\Desafio14-Analisis-De-Performance
- Memoria Total Reservada (rss): 71749632
- Cantidad de procesadores: 4



Name	Status	Type	Initiator	Size
info	200	document	Other	1.2 KB
pico.min.css	302	stylesheet / ...	info	145 B
style.css	200	stylesheet	info	1.3 KB
normalizr.brower.min.js	200	script	info	2.6 KB
socket.io.js	200	script	info	26.0 KB
handlebars.min.js	200	script	info	20.2 KB
scripts.js	200	script	info	5.8 KB
pico.min.css	200	stylesheet	pico.min.css	10.7 KB
?EIO=4&transport=polling&tt=OKoGeDL	200	xhr	polling.js:311	283 B
?EIO=4&transport=polling&tt=OKoGeDf...	200	xhr	polling.js:311	149 B
?EIO=4&transport=websocket&sid=y6r...	101	websocket	websocket.js:54	0 B
?EIO=4&transport=polling&tt=OKoGeE3...	200	xhr	polling.js:311	318 B

Figura 2. Captura de pantalla de cantidad de bytes con compresión

2. Vamos a trabajar sobre la ruta '/info', en modo fork, agregando ó extrayendo un console.log de la información colectada antes de devolverla al cliente. Además, desactivaremos el child_process de la ruta '/randoms'.

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

- a. El perfilamiento del servidor, realizando el test con --prof de node.js. Analizar los resultados obtenidos luego de procesarlos con --prof-process.

Utilizaremos como test de carga Artillery en línea de comandos, emulando 50 conexiones concurrentes con 20 request por cada una. Extraer un reporte con los resultados en archivo de texto.

- b. El perfilamiento del servidor con el modo inspector de node.js --inspect. Revisar el tiempo de los procesos menos performantes sobre el archivo fuente de inspección.

2.a. Para empezar, agrego un console.log de la información colectada en la ruta /info y prendo el servidor en modo profiler con el siguiente comando:

```
npm run prof
```

Posteriormente ejecuto el test de carga de Artillery con el siguiente comando:

```
artillery quick --count 20 -n 50 "http://localhost:8080/info" > result_con_consoleLog.txt
```

Termino la ejecución del servidor, elimino el console.log y vuelvo a prender el servidor en modo profiler, pero ahora para ejecutar el test de carga utilizo la siguiente línea de comando

```
artillery quick --count 20 -n 50 "http://localhost:8080/info" > result_sin_consoleLog.txt
```

Posteriormente, descripto los archivos de log que se crearon con el modo profiler, usando los siguientes comandos:

```
node --prof-process isolate-con-consolelog-v8 > result_con_consoleLog_v8.txt
```

```
node --prof-process isolate-sin-consolelog-v8 > result_sin_consoleLog_v8.txt
```

Los archivos generados se encuentran ubicados dentro de la carpeta test/artillery.

Si analizamos el reporte generado por Archillery (Figura 3 y Figura 4) podemos comprobar que los tiempos de respuestas fueron considerablemente superiores cuando se utilizó console.log. Por otro lado, si analizamos los resultados obtenidos del perfilamiento del servidor podemos comprobar que sin usar console.log tenemos aproximadamente 5000 ticks menos.

```

-----
Summary report @ 01:55:32(-0300)
-----

http.codes.200: ..... 1000
http.request_rate: ..... 68/sec
http.requests: ..... 1000
http.response_time:
  min: ..... 15
  max: ..... 414
  median: ..... 247.2
  p95: ..... 368.8
  p99: ..... 399.5
http.responses: ..... 1000
vusers.completed: ..... 20
vusers.created: ..... 20
vusers.created_by_name.0: ..... 20
vusers.failed: ..... 0
vusers.session_length:
  min: ..... 13419.4
  max: ..... 13957.4
  median: ..... 13770.3
  p95: ..... 14048.5
  p99: ..... 14048.5

```

Figura 3. Reporte de Artillery utilizando console.log

```

-----
Summary report @ 02:22:59(-0300)
-----

http.codes.200: ..... 1000
http.request_rate: ..... 98/sec
http.requests: ..... 1000
http.response_time:
  min: ..... 23
  max: ..... 318
  median: ..... 149.9
  p95: ..... 252.2
  p99: ..... 273.2
http.responses: ..... 1000
vusers.completed: ..... 20
vusers.created: ..... 20
vusers.created_by_name.0: ..... 20
vusers.failed: ..... 0
vusers.session_length:
  min: ..... 8112.3
  max: ..... 8653.1
  median: ..... 8520.7
  p95: ..... 8520.7
  p99: ..... 8520.7

```

Figura 4. Reporte de Artillery sin utilizar console.log

[Summary]:				[Summary]:			
ticks	total	nonlib	name	ticks	total	nonlib	name
61	0.4%	100.0%	JavaScript	96	0.9%	99.0%	JavaScript
0	0.0%	0.0%	C++	0	0.0%	0.0%	C++
46	0.3%	75.4%	GC	38	0.4%	39.2%	GC
15180	99.6%		Shared libraries	10142	99.1%		Shared libraries
				1	0.0%		Unaccounted

Figura 5. Resumen del perfilamiento del servidor con console.log (Izquierda) y sin console.log (Derecha)

2.b. Realizando el perfilamiento del servidor con node inspect se obtuvieron los tiempos que se muestran en la Figura 6. Claramente utilizando console.log hay un aumento considerable de los tiempos.

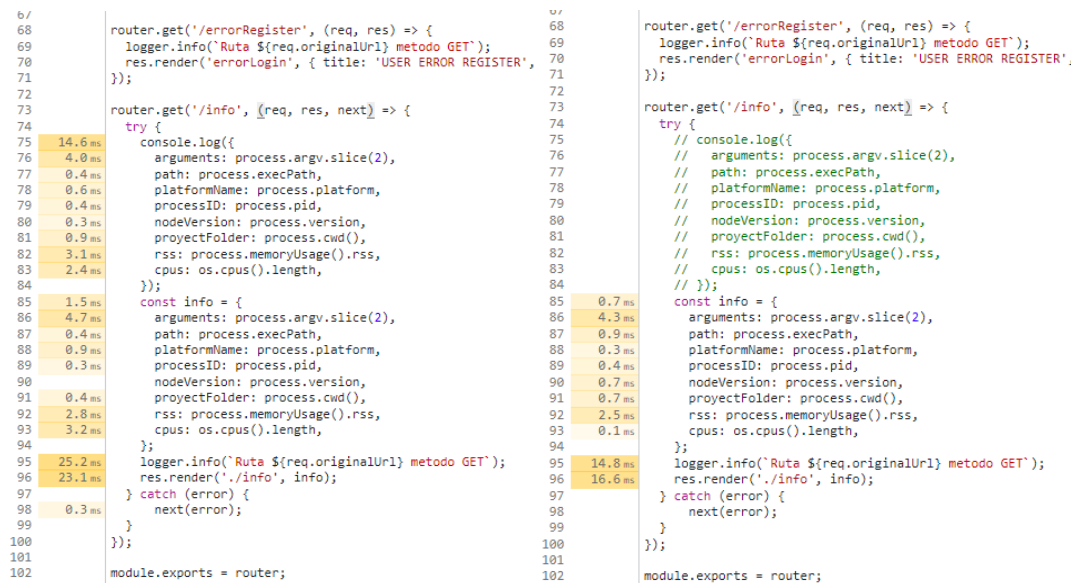


Figura 6. Tiempos de procesos usando node inspect. Con console.log (Izquierda) y sin console.log (Derecha)

- Luego utilizaremos Autocannon en línea de comandos, emulando 100 conexiones concurrentes realizadas en un tiempo de 20 segundos. Extraer un reporte con los resultados (puede ser un print screen de la consola)
- El diagrama de flama con Ox, emulando la carga con Autocannon con los mismos parámetros anteriores.

En las Figuras 7 y 8 se muestran los diagramas de flama obtenidos con Ox. En ellos prácticamente no se logra diferenciar la carga con console.log y sin console.log



Figura 7. Diagrama de flama de Ox con console.log

node app.js



Figura 8. Diagrama de flama sin console.log