

CODERHOUSE - ENTREGA 16

ANÁLISIS COMPLETO DE PERFORMANCE

Aclaración: trabajo realizado sobre el endpoint `/api/env/info`, variando entre información BLOQUEANTE y NO BLOQUEANTE.

1) Perfilamiento de servidor (profiler)

ARTILLERY:

Tipo de test de carga por consola.

NO BLOQUEANTE:

1º Ejecutamos el servidor con: `node --prof src/index.js`

2º En otra consola, ejecutamos la prueba de 20 request en 50 cuentas con artillery :
`artillery quick --count 20 -n 50 "http://localhost:8080/api/env/info" > result_nobloq.txt`

Esto nos arroja el siguiente resultado:

```
-----
Summary report @ 10:41:57(-0300)
-----

http.codes.200: ..... 1000
http.request_rate: ..... 306/sec
http.requests: ..... 1000
http.response_time:
  min: ..... 1
  max: ..... 43
  median: ..... 15
  p95: ..... 23.8
  p99: ..... 32.1
http.responses: ..... 1000
vusers.completed: ..... 20
vusers.created: ..... 20
vusers.created_by_name.0: ..... 20
vusers.failed: ..... 0
vusers.session_length:
  min: ..... 402.9
  max: ..... 936.4
  median: ..... 820.7
  p95: ..... 925.4
  p99: ..... 925.4
```

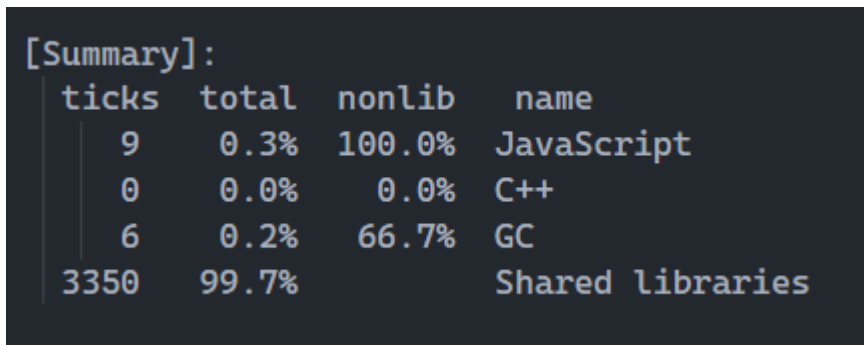
3° Apagamos el servidor

4° Renombramos el archivo isolate que se crea al iniciar el servidor en modo profiler a "nobloq-v8".

log"

5° Ejecutamos: `node.exe --prof-process nobloq-v8.log > result_nobloq-v8.txt`. Esto procesa el archivo .log a uno .txt con formato resumido.

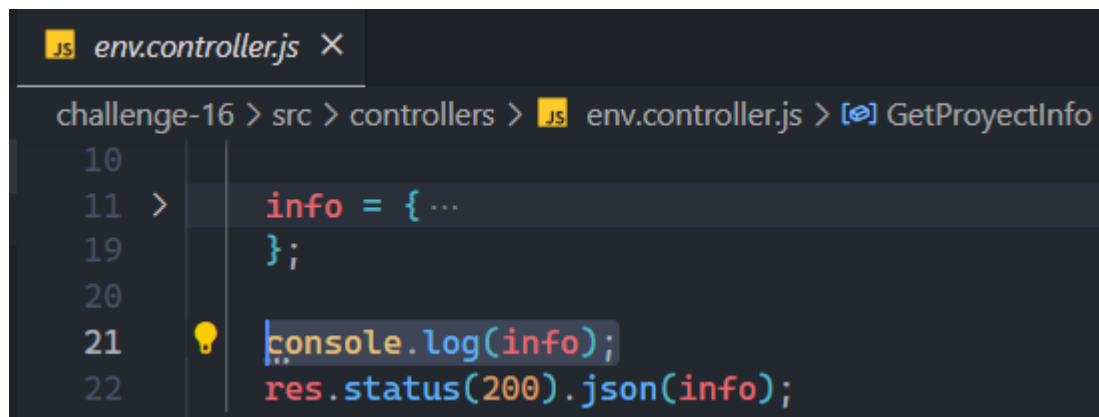
Nos arroja el siguiente resultado:



[Summary]:			
ticks	total	nonlib	name
9	0.3%	100.0%	JavaScript
0	0.0%	0.0%	C++
6	0.2%	66.7%	GC
3350	99.7%		Shared libraries

BLOQUEANTE:

1° En el método que envía la información relevante, incluimos un `console.log` con la misma antes de que se realice el envío:



```
challenge-16 > src > controllers > JS env.controller.js > [?] GetProyectInfo
10
11 > info = { ...
19   };
20
21  console.log(info);
22  res.status(200).json(info);
```

2° Ejecutamos el servidor con: `node --prof src/index.js`

3° En otra consola, ejecutamos la prueba de 20 request en 50 cuentas con artillery :
`artillery quick --count 20 -n 50 "http://localhost:8080/api/env/info" > result_nobloq.txt`

Esto nos arroja el siguiente resultado:

```
-----  
Summary report @ 10:46:39(-0300)  
-----
```

```
http.codes.200: ..... 1000  
http.request_rate: ..... 165/sec  
http.requests: ..... 1000  
http.response_time:  
  min: ..... 4  
  max: ..... 86  
  median: ..... 45.2  
  p95: ..... 66  
  p99: ..... 82.3  
http.responses: ..... 1000  
vusers.completed: ..... 20  
vusers.created: ..... 20  
vusers.created_by_name.0: ..... 20  
vusers.failed: ..... 0  
vusers.session_length:  
  min: ..... 1793.3  
  max: ..... 2419.3  
  median: ..... 2276.1  
  p95: ..... 2416.8  
  p99: ..... 2416.8
```

3° Apagamos el servidor

4° Renombramos el archivo isolate que se crea al iniciar el servidor en modo profiler a "bloq-v8".

log"

5° Ejecutamos: *node.exe --prof-process bloq-v8.log > result_bloq-v8.txt*. Esto procesa el archivo .log a uno .txt con formato resumido.

Nos arroja el siguiente resultado:

```
[Summary]:  
  ticks  total  nonlib   name  
    9     0.2%  100.0%  JavaScript  
    0     0.0%   0.0%    C++  
    3     0.1%  33.3%    GC  
  5090    99.8%           Shared libraries
```

AUTOCANNON:

Tipo de test de carga por código.

Se instala a nivel de proyecto con *npm install autocannon*.

Creamos un benchmark y lo configuramos de la siguiente manera:

```
JS benchmark.js X
challenge-16 > tests > performance > autocannon > JS benchmark.js > ...
1  const autocannon = require("autocannon");
2  const { PassThrough } = require("stream");
3
   Complexity is 3 Everything is cool!
4  function run(url) {
5      const buf = [];
6      const outputStream = new PassThrough();
7
8      const inst = autocannon({
9          url,
10         connections: 100,
11         duration: 20,
12     });
13
14     autocannon.track(inst, { outputStream });
15
16     outputStream.on("data", (data) => buf.push(data));
17     inst.on("done", function () {
18         process.stdout.write(Buffer.concat(buf));
19     });
20 }
21
22 console.log("Running all benchmark in parallel...");
23
24 run("http://localhost:8080/api/env/info");
```

NO BLOQUEANTE:

1º Iniciamos el servidor con: *node --prof src/index.js*

2º En otra consola, ejecutamos el test de autocannon con *npm run test*. (He creado el script para el mismo)

3º Obtenemos en consola el siguiente resultado:

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

Stat	2.5%	50%	97.5%	99%	Avg	Stdev	Max
Latency	87 ms	97 ms	158 ms	190 ms	107.19 ms	22.89 ms	279 ms

Stat	1%	2.5%	50%	97.5%	Avg	Stdev	Min
Req/Sec	625	625	984	1040	926.55	123.55	625
Bytes/Sec	414 kB	414 kB	651 kB	687 kB	612 kB	81.6 kB	413 kB

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

19k requests in 20.05s, 12.2 MB read
```

BLOQUEANTE:

- 1º En el método que envía la información relevante, incluimos un `console.log` con la misma antes de que se realice el envío.
- 2º Iniciamos el servidor con: `node --prof src/index.js`
- 3º En otra consola, ejecutamos el test de autocannon con `npm run test`. (He creado el script para el mismo)
- 4º Obtenemos en consola el siguiente resultado:

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

Stat	2.5%	50%	97.5%	99%	Avg	Stdev	Max
Latency	231 ms	256 ms	302 ms	309 ms	257.12 ms	22.56 ms	343 ms

Stat	1%	2.5%	50%	97.5%	Avg	Stdev	Min
Req/Sec	330	330	395	418	386.15	21.13	330
Bytes/Sec	218 kB	218 kB	261 kB	276 kB	255 kB	14 kB	218 kB

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

8k requests in 20.07s, 5.11 MB read
```

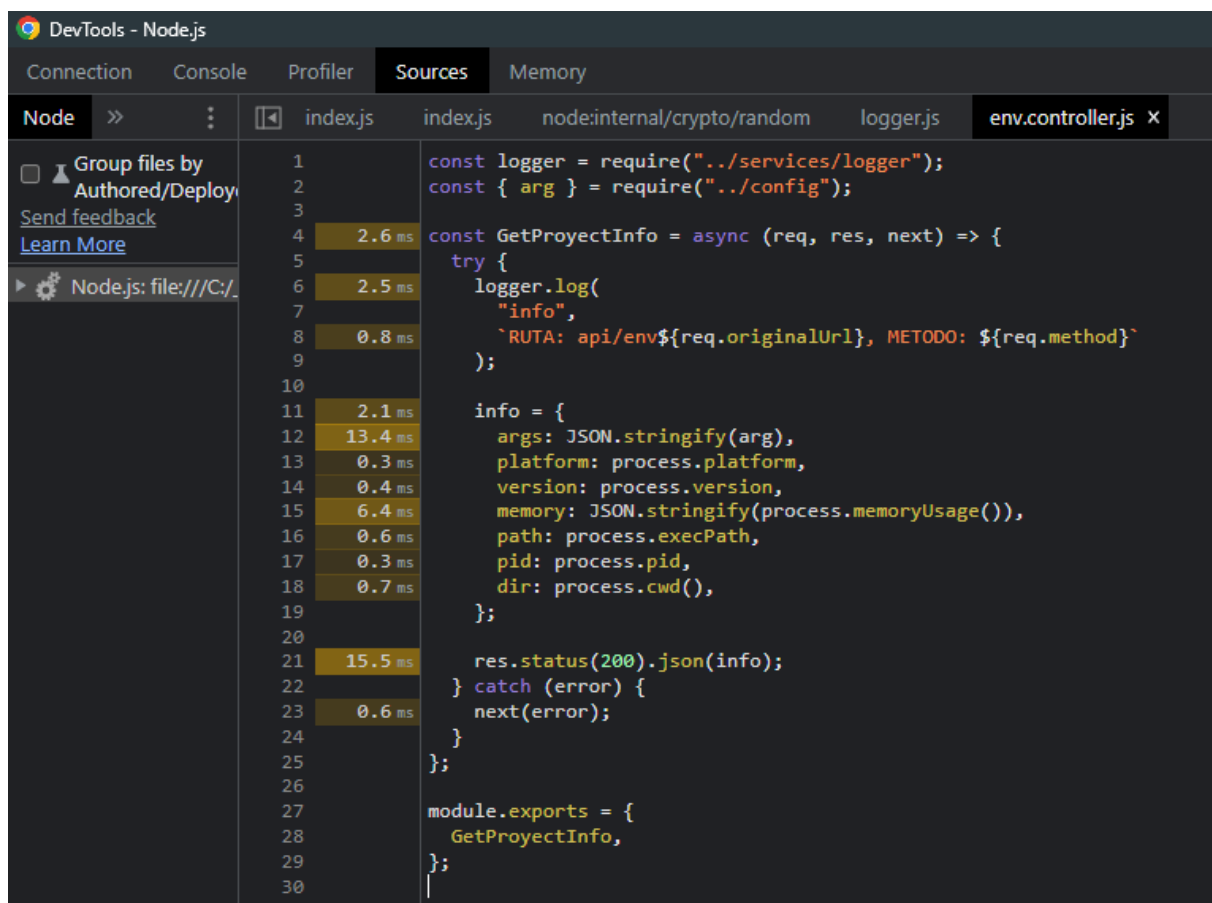
2) Perfilamiento de servidor en modo Inspect

NO BLOQUEANTE:

1º Ejecutamos el servidor con: `node --inspect src/index.js`

2º En otra consola, ejecutamos la prueba de 20 request en 50 cuentas con artillery :
`artillery quick --count 20 -n 50 "http://localhost:8080/api/env/info"`

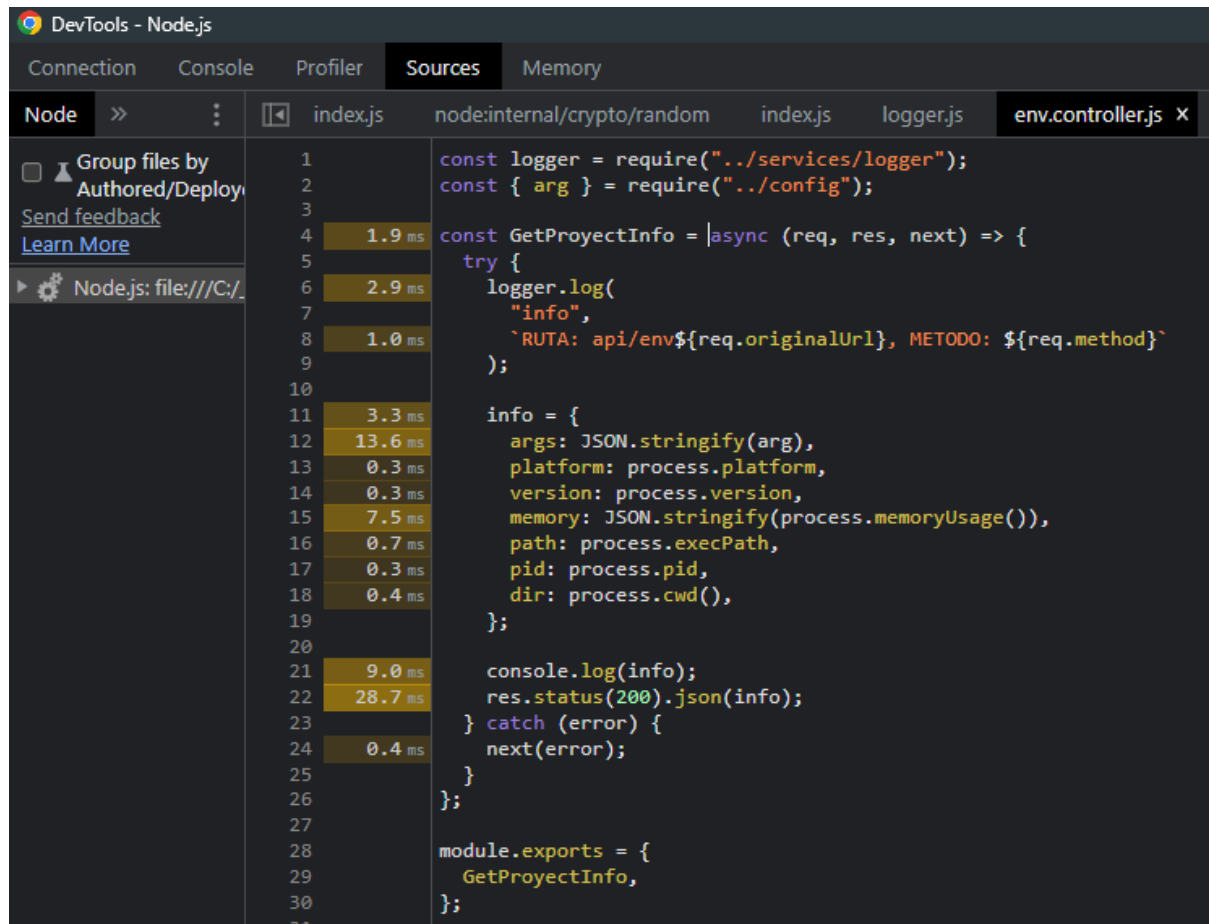
3º Abrimos las DevTools, menú Profiler y buscando entre los logs, abrimos lo que nos parezca relevante de verificar, en este caso el controlador que maneja el envío de la info:



BLOQUEANTE:

Mismos pasos anteriores pero recordando agregar el `console.log` en el controlador, antes de iniciar el servidor.

Inspeccionando DevTools nos encontramos:



3) Diagrama de Flama usando 0x

0x es un perfilador y generador de gráficos interactivos llamados “flama” para procesos Node relevantes.

Usaremos Autocannon para el test de carga.

NO BLOQUEANTE:

1º Ejecutamos el servidor en modo 0x con: *node run dev-0x*. (He creado el script para el mismo)

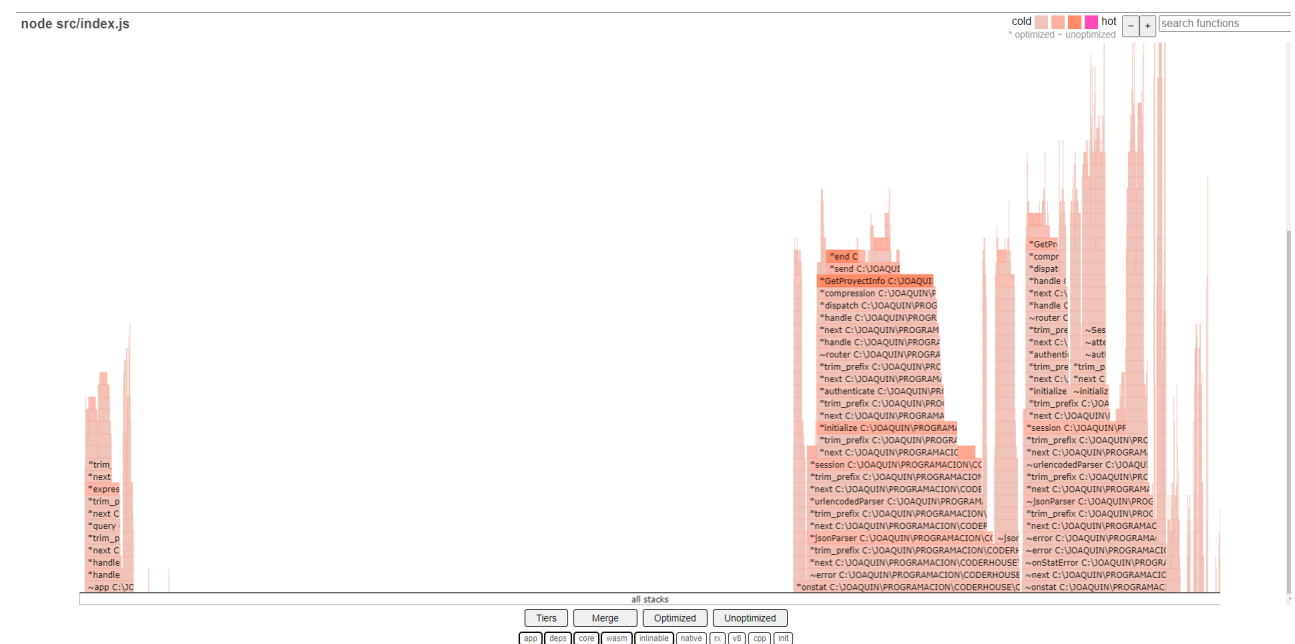
2º En otra consola, ejecutamos el test de autocannon con *npm run test*.

3º Una vez terminado el proceso de autocannon, cerramos el servidor con CTRL + C apretandolo **1 SOLA VEZ**, esperamos que genera el html del gráfico y volvemos a apretar CTRL + C para confirmar el apagado del servidor.

4º Buscamos el html del gráfico generado, el mismo se genera en una carpeta random con varios archivos aparte.

[illegible]

El gráfico obtenido es el siguiente:



CONCLUSIÓN

Con respecto a los resultados obtenidos, basándonos en **ARTILLERY**, podemos analizar que el servidor NO BLOQUEANTE cuenta con aproximadamente el 50% de ticks menos que el servidor BLOQUEANTE. En cuanto a velocidad de respuesta, el servidor NO BLOQUEANTE respondió con una velocidad de 306/sec y el BLOQUEANTE con 165/sec, es decir, aproximadamente la mitad, lo cual genera una velocidad de respuesta promedio del triple de lenta que el servidor NO BLOQUEANTE.

Basándonos ahora en las pruebas obtenidas con **AUTOCANNON**, podemos ver como el servidor NO BLOQUEANTE, es aproximadamente un 250% más rápido, cuenta con menor latencia y con mayor respuesta por segundo.

Como conclusión general podemos tener en claro que cualquier proceso que tenga que realizarse de forma SINCRÓNICA, como lo es el console.log utilizado para las pruebas, puede generar un efecto negativo importante en nuestra aplicación.