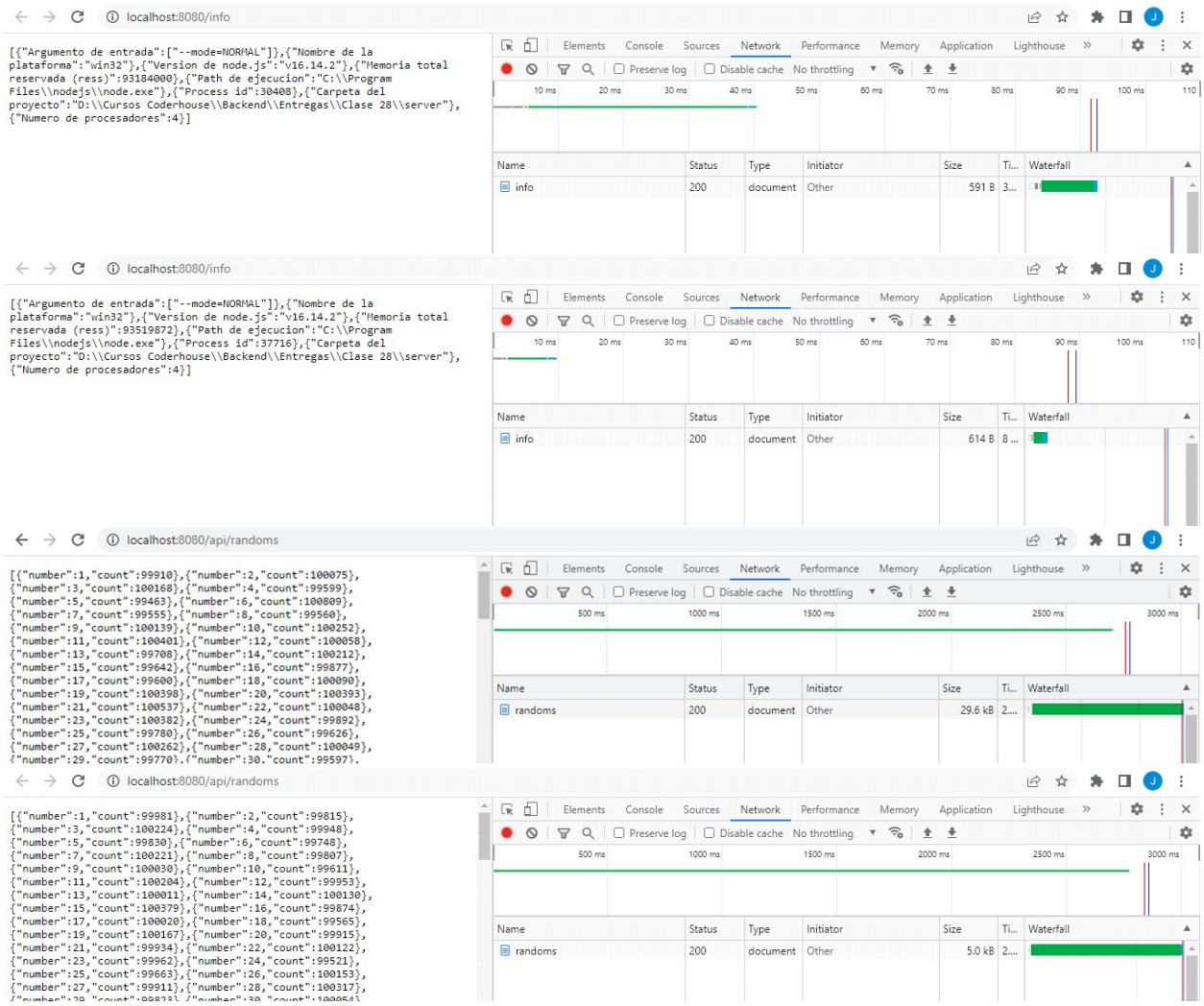
1. Análisis de Compresión

Se hizo la prueba de compresión para la ruta /info y /randoms

Dado que la ruta /info solo devuelve un json, la compresión generó un archivo 23 bytes más pesado (591B vs 614B).

Al probar con más volumen de datos, en la ruta /randoms la compresión fue significativa. Sin compresión pesó 29.6kB vs 5kB con compresión.



2. Análisis de rendimiento con node profiler + artillery

2. a. Análisis de rendimiento con console.log

Pasos:

1. Agregar console.log(data) antes de enviar la respuesta de data en /info en el archivo infoRoutes.js
2. Iniciar el servidor en modo profiler

```
node --prof src/server.js
```

3. Abrir otra terminal y correr artillery

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

4. Detener el servidor
5. Renombrar el archivo .log generado a

```
prof-conclg.log
```

6. Traducir los resultados del profiler de node

```
node --prof-process prof-conclg.log > res-prof-conclg.txt
```

2. b. Análisis de rendimiento sin console.log

Pasos:

1. Dejar comentada la línea de console.log(data) que esta antes de enviar la respuesta de data en /info en el archivo infoRoutes.js
2. Iniciar el servidor en modo profiler

```
node --prof src/server.js
```

3. Abrir otra terminal y correr artillery

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

4. Detener el servidor
5. Renombrar el archive .log generado a

```
prof-sinclg.log
```

6. Traducir los resultados del profiler de node

```
node --prof-process prof-sinclg.log > res-prof-sinclg.txt
```

2. c. Conclusiones

Resumen de resultados de profiling con console.log

```
[Summary]:
   ticks  total  nonlib   name
     10    0.2%  100.0%  JavaScript
      0    0.0%    0.0%  C++
     17    0.4%  170.0%  GC
   4761   99.8%          Shared libraries
```

Resumen de resultados de profiling sin console.log

```
[Summary]:
   ticks  total  nonlib   name
     10    0.4%  100.0%  JavaScript
      0    0.0%    0.0%  C++
     14    0.5%  140.0%  GC
   2546   99.6%          Shared libraries
```

Al comparar se puede concluir que la cantidad de ticks que requiere el servidor con console.log es casi el doble que sin console.log. Esto hace que el proceso con loggeo de consola sea mucho más lento.

Analizando los resultados de aritllery, también se puede concluir o mismo.

Resultados de artillery con console.log

```
http.codes.200: ............................................................ 1000
http.request_rate: ......................................................... 67/sec
http.requests: ............................................................. 1000
```

Resultados de artillery sin console.log

```
http.codes.200: ............................................................ 1000
http.request_rate: ......................................................... 166/sec
http.requests: ............................................................. 1000
```

Si bien para ambos casos se pudieron atender todas las solicitudes, la cantidad de solicitudes por segundo que atiende el servidor sin console.log es casi el triple que con console.log

3. Análisis de rendimiento con node inspect + autocannon

3. a. Análisis de rendimiento con console.log

Pasos:

1. Agregar console.log(data) antes de enviar la respuesta de data en /info en el archivo infoRoutes.js
2. Iniciar el servidor en modo inspect

```
node --inspect src/server.js
```

3. Abrir Chrome y poner la siguiente url

```
chrome://inspect
```

4. Click en Open dedicated DevTools for Node
5. Ir a la solapa profiler y apretar el boton Start para comenzar a grabar con el profiler de Chrome
6. Correr benchmark.js con autocannon

```
node src/benchmark.js
```

7. Una vez finalizado detener el profiler

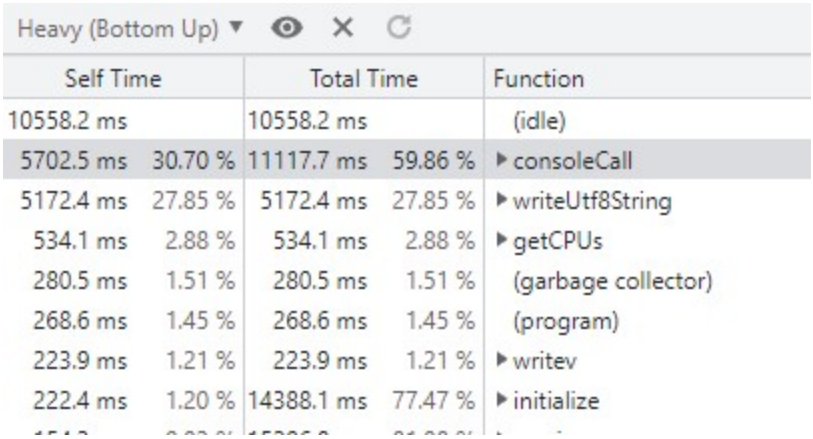
3. b. Análisis de rendimiento sin console.log

Los pasos son exactamente los mismos, solo que comentando la línea de console.log en el archivo infoRoutes.js

3. c. Conclusiones

Resultados con console.log

Al ver los resultados con console.log se puede notar que la instrucción consoleCall es la que más tiempo consume del proceso, generando un cuello de botella.



Self Time		Total Time		Function
10558.2 ms		10558.2 ms		(idle)
5702.5 ms	30.70 %	11117.7 ms	59.86 %	consoleCall
5172.4 ms	27.85 %	5172.4 ms	27.85 %	writeUtf8String
534.1 ms	2.88 %	534.1 ms	2.88 %	getCPUs
280.5 ms	1.51 %	280.5 ms	1.51 %	(garbage collector)
268.6 ms	1.45 %	268.6 ms	1.45 %	(program)
223.9 ms	1.21 %	223.9 ms	1.21 %	writev
222.4 ms	1.20 %	14388.1 ms	77.47 %	initialize

```
1                 import express from "express";
2                 const router = express.Router();
3                 import os from "os";
4
5     0.5 ms      export default router.get("/", (req, res) => {
6     0.5 ms        const data = [];
7     8.5 ms        data.push({ "Argumento de entrada": process.argv.slice(2) });
8     1.0 ms        data.push({ "Nombre de la plataforma": process.platform });
9     1.3 ms        data.push({ "Version de node.js": process.version });
10    2.6 ms        data.push({ "Memoria total reservada (ress)": process.memoryUsage.rss() });
11    1.6 ms        data.push({ "Path de ejecucion": process.argv[0] });
12    1.6 ms        data.push({ "Process id": process.pid });
13    1.1 ms        data.push({ "Carpeta del proyecto": process.cwd() });
14    7.3 ms        data.push({ "Numero de procesadores": os.cpus().length });
15                  // Activo o desactivo el console.log de la data
16   20.6 ms        console.log(data)
17   39.5 ms        res.send(data);
18                });
19
```

Resultados sin console.log

Al ver los resultados sin console.log se puede notar que la instrucción consoleCall ya no está y esto mejora de gran manera el rendimiento.

Self Time		Total Time		Function
12935.4 ms		12935.4 ms		(idle)
2265.5 ms	12.04 %	2265.5 ms	12.04 %	▸ writeUtf8String
1714.0 ms	9.11 %	1714.0 ms	9.11 %	▸ getCPUs
731.0 ms	3.89 %	731.0 ms	3.89 %	▸ writev
711.0 ms	3.78 %	8552.2 ms	45.47 %	▸ initialize
679.8 ms	3.61 %	679.8 ms	3.61 %	(program)
401.4 ms	2.13 %	401.4 ms	2.13 %	(garbage collector)
398.9 ms	2.12 %	76833.6 ms	408.47 %	▸ next
372.0 ms	1.98 %	641.0 ms	3.41 %	▸ nextTick
364.2 ms	1.94 %	10657.3 ms	56.66 %	▸ session
306.6 ms	1.63 %	695.4 ms	3.70 %	▸ hash
290.4 ms	1.54 %	1147.1 ms	6.10 %	▸ compression
251.1 ms	1.33 %	9312.1 ms	49.51 %	▸ send
236.3 ms	1.26 %	236.3 ms	1.26 %	▸ Hash

infoRoutes.js × node:internal/crypto/pbkdf2

```
1                 import express from "express";
2                 const router = express.Router();
3                 import os from "os";
4
5       1.3 ms    export default router.get("/", (req, res) => {
6       0.3 ms      const data = [];
7      23.6 ms      data.push({ "Argumento de entrada": process.argv.slice(2) });
8       1.8 ms      data.push({ "Nombre de la plataforma": process.platform });
9       2.8 ms      data.push({ "Version de node.js": process.version });
10      5.7 ms      data.push({ "Memoria total reservada (ress)": process.memoryUsage.rss() });
11      2.8 ms      data.push({ "Path de ejecucion": process.argv[0] });
12      3.4 ms      data.push({ "Process id": process.pid });
13      4.1 ms      data.push({ "Carpeta del proyecto": process.cwd() });
14      5.7 ms      data.push({ "Numero de procesadores": os.cpus().length });
15                  // Activo o desactivo el console.log de la data
16                  // console.log(data)
17    122.5 ms      res.send(data);
18      0.4 ms    });
19
```

4. Análisis de rendimiento con 0x + autocannon

4. a. Análisis de rendimiento con console.log

Pasos:

1. Agregar console.log(data) antes de enviar la respuesta de data en /info en el archivo infoRoutes.js
2. Iniciar el servidor on 0x

```
0x src/server.js
```

3. Correr benchmark.js con autocannon

```
node src/benchmark.js
```

4. Una vez finalizado detener el servidor, se generará automáticamente una carpeta con el diagrama de flama. Renombrar esta carpeta para poder identificarla luego:

```
0x-con-clg
```

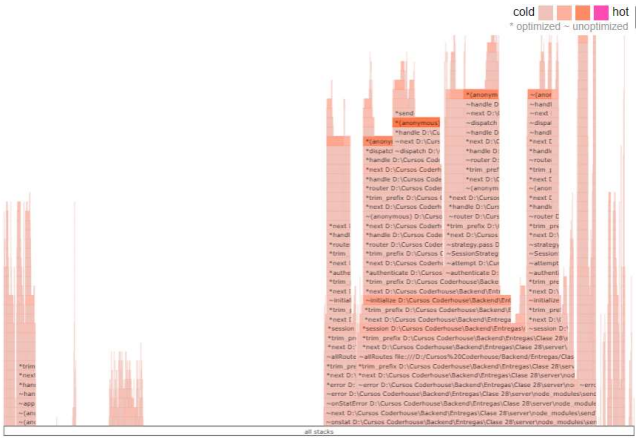
4. b. Análisis de rendimiento sin console.log

Los pasos son exactamente los mismos, solo que comentando la línea de console.log en el archivo infoRoutes.js. Renombrar la carpeta creada para poder identificarla luego:

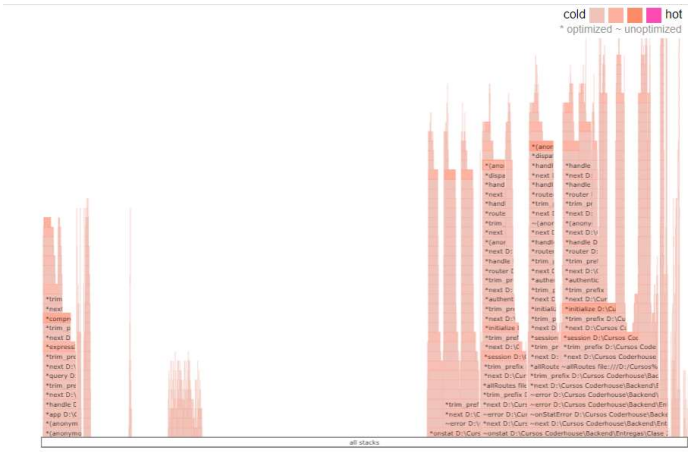
```
0x-sin-clg
```

4. c. Conclusiones

Resultados con console.log



Resultados sin console.log



Al comparar ambos gráficos se puede concluir que si bien infoRoutes en ambos casos es la ruta que más exige al servidor (debido a la cantidad de requests) el diagrama de flama, tanto en su espaciado como en su color (más extendido y más oscuro en el caso de la consola), se puede concluir que el código con consola demora más el proceso.