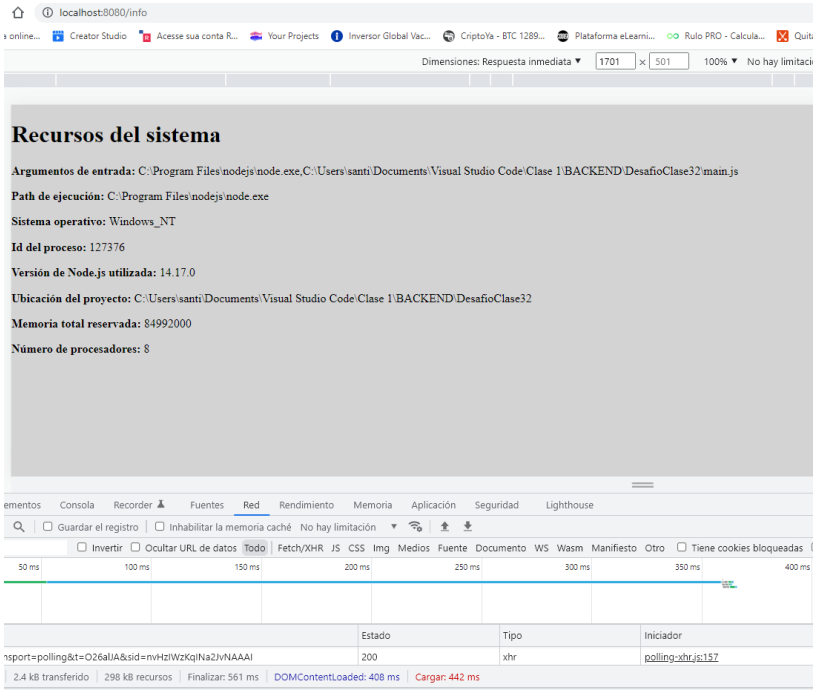
ANÁLISIS DE RENDIMIENTO

Uso de Gzip

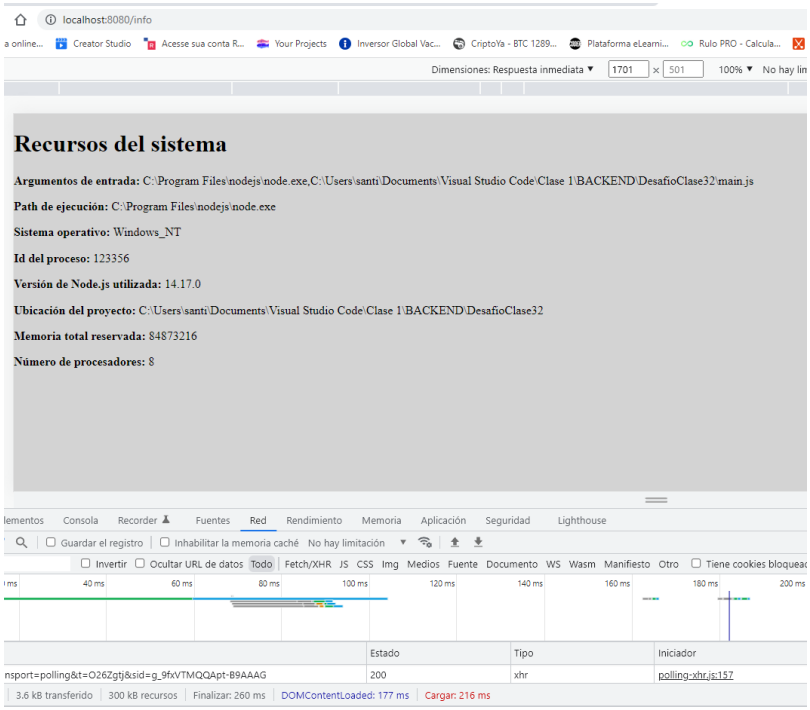
Se comparó la diferencia entre usar la función compression o no en el código y se midió en el análisis de red del navegador la cantidad de bytes recibidos:

```
app.use(compression())
```

Con compresión



Sin compresión



En el primer caso (con compresión) el navegador recibió 2,4kB de datos y en el segundo, 3,6kB. Esto resultó en un ahorro de datos del 33,3%

Análisis de performance

Se desactivó el proceso fork() en la ruta /api/randoms y se trabajó sobre la ruta /info. Se hicieron dos análisis de performance para analizar qué sucede si se intercala en el código una función console.log como se muestra más abajo. Se usó Artillery emulando 50 conexiones con 20 request cada una, y Node built-in profiler para procesar los datos recibidos.

```
console.log (   `layout: 'info',
                rss:${process.memoryUsage().rss.toString()},
                argv: ${process.argv},
                cwd: ${process.cwd()},
                nodeVersion: ${process.env.npm_config_node_version},
                execPath: ${process.execPath},
                versionSO: ${process.env.OS},
                pid: ${process.pid.toString()},
                cpus: ${CPUS}`
                )
```

En la sección Summary se obtuvieron los siguientes resultados:

Con console.log	Sin console.log
[Summary]: ticks total nonlib name 33 0.9% 97.1% JavaScript 0 0.0% 0.0% C++ 22 0.6% 64.7% GC 3445 99.0% Shared libraries 1 0.0% Unaccounted	[Summary]: ticks total nonlib name 39 0.9% 100.0% JavaScript 0 0.0% 0.0% C++ 30 0.7% 76.9% GC 4288 99.1% Shared libraries

Se concluye que, en nuestro caso, el proceso que ejecuta el console.log tiene muchos menos ticks en Shared libraries que en el proceso que no lo ejecuta. Se supone que esta función es bloqueante, por lo que debería ser al revés. Habría que probar en un número mucho mayor de conexiones y request para ver si las proporciones se mantienen.

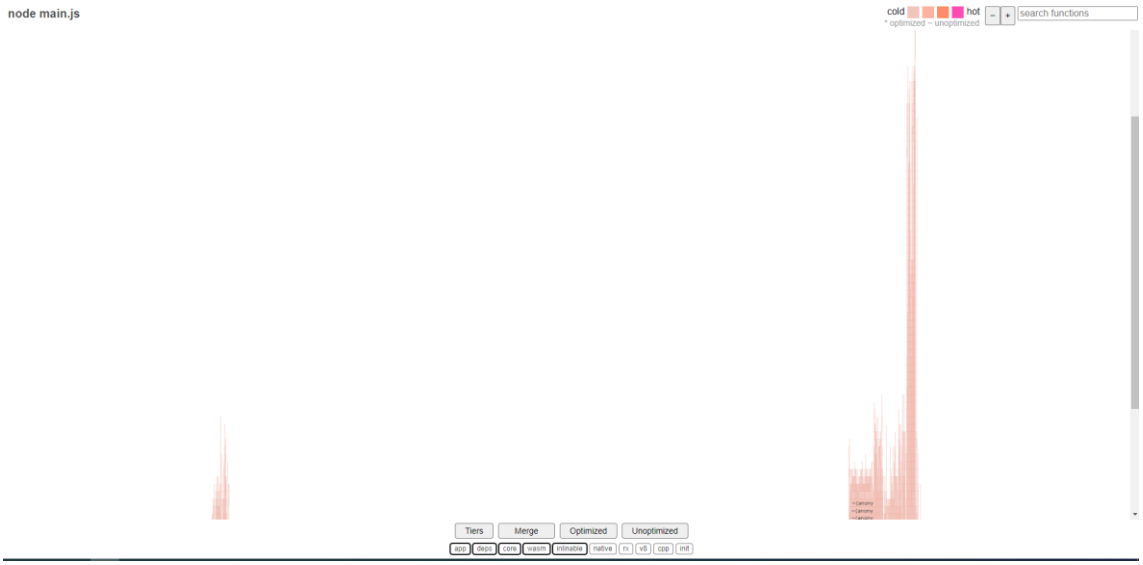
Test de carga

Se realizó con Autocannon y 0x para luego realizar un mapa de calor del servidor. Esto fue lo que devolvió Autocannon

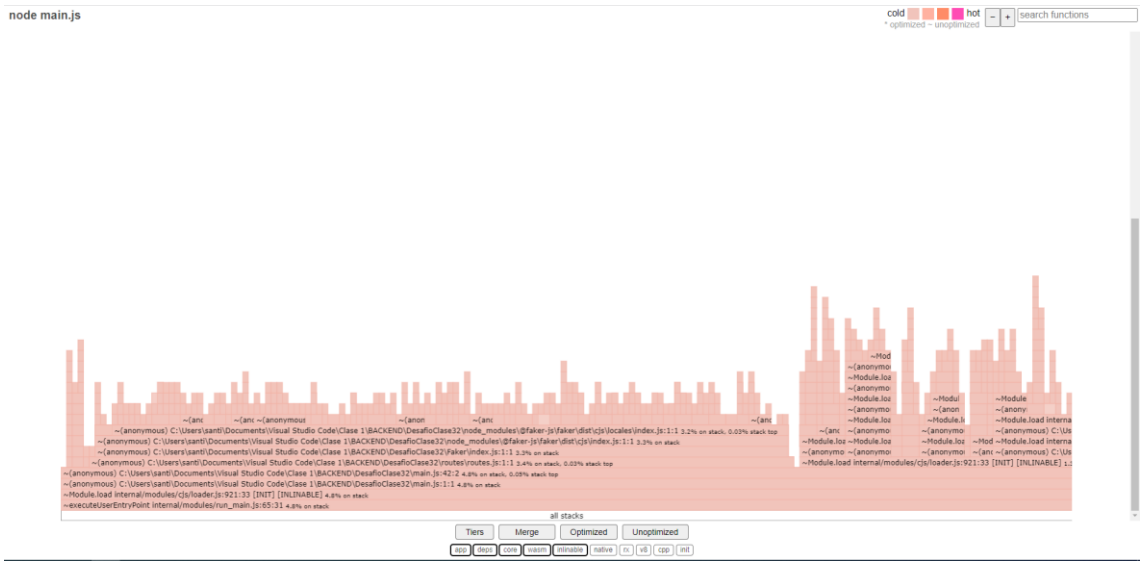
Stat	2.5%	50%	97.5%	99%	Avg	Stdev	Max
Latency	702 ms	1079 ms	3443 ms	3873 ms	1248.03 ms	605.83 ms	4167 ms

Stat	1%	2.5%	50%	97.5%	Avg	Stdev	Min
Req/Sec	6	6	84	131	78.6	32.52	6
Bytes/Sec	10.2 kB	10.2 kB	143 kB	223 kB	134 kB	55.4 kB	10.2 kB

Y luego el mapa de calor obtenido con 0x:



Y haciendo un Zoom:



Conclusiones:

-GZip en efecto reduce la cantidad de Bytes a enviar al navegador. Sin embargo, en nuestro caso se puede ver que prácticamente se duplica el tiempo de carga al usar compresión. También hay que tener en cuenta que, según los apuntes de clase, este método no es recomendado cuando hay mucho tráfico.

- A pesar de los resultados obtenidos por Artillery en nuestro análisis de performance, los porcentajes de ticks recibidos en los dos tipos de test (con y sin función bloqueante) son similares. Habría que probar qué resultados se obtienen si se intercalan más funciones bloqueantes, y si en efecto se llega a la conclusión lógica de que las funciones no bloqueantes reducen los tiempos de respuesta del servidor.

- Y finalmente, se pudo hacer funcionar el “mapa de calor” que se obtiene combinando el uso de Autocannon con 0x. En nuestro código a analizar no hay muchas funciones bloqueantes pues no hay muchos console.log y siempre se fue trabajando con funciones asíncronas a lo largo del curso.