Desafío entregable Clase 32: Curso de Backend – CoderHouse

Compresión con GZIP

Vamos a ver en el siguiente ejemplo como se mejora el rendimiento en producción gracias a la compresión con GZIP.

En los siguientes ejemplos al ser la información menor a 2kB, no se vio una mejoría con la compresión con GZIP.

Bibliografía web al respecto:

https://www.humanlevel.com/en/wpo/gzip-compression-from-top-to-bottom.html

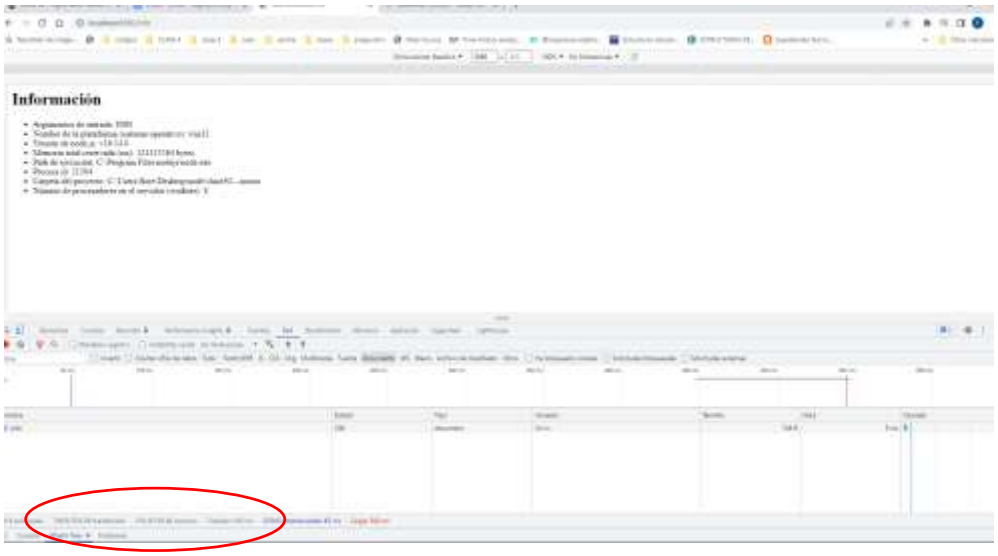
We should also consider the file size. It's not worth compressing a file smaller than 1KB, because it is likely that when it's sent over the network, the information will never be segmented, taking the same time to be transmitted either compressed or not, with the added cost for the CPU of having to work with compressed information in the client and the server. It's not easy to establish what should be the minimum size for using compression, because it depends on various factors of the networks it has to go through to reach its destination. Generally speaking, it's usually worse for performance if we compress files below 1 or 2KB.

https://www.quora.com/Can-gzip-file-compression-increase-the-file-size

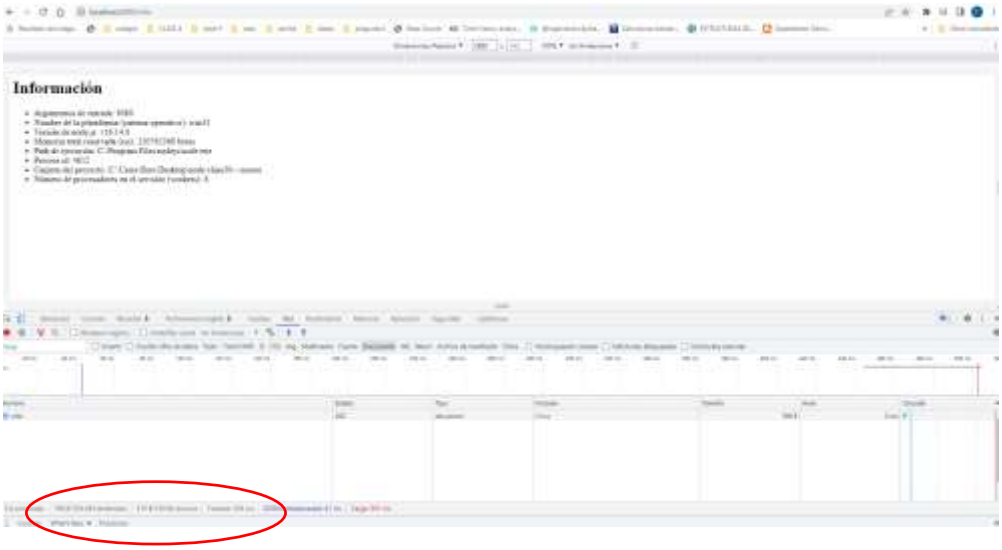
You are right Gzip smaller file will increase the file size. Due to the overhead and latency of compression and decompression, gzip is not useful for a files smaller than 1000 bytes. Gzipping files below 1000 bytes can actually make them larger

Vemos igualmente los resultados

SIN GZIP – Tamaño 746B, Tiempo 545mseg



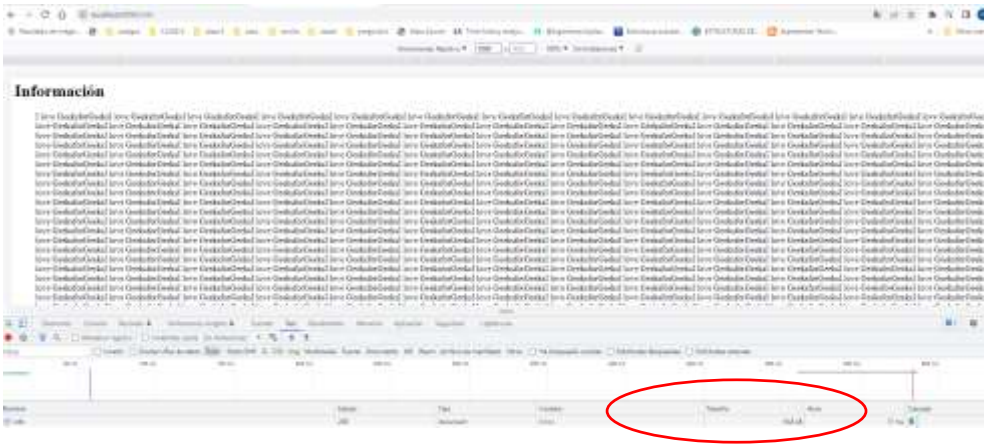
CON GZIP– Tamaño 768B, Tiempo 503mseg



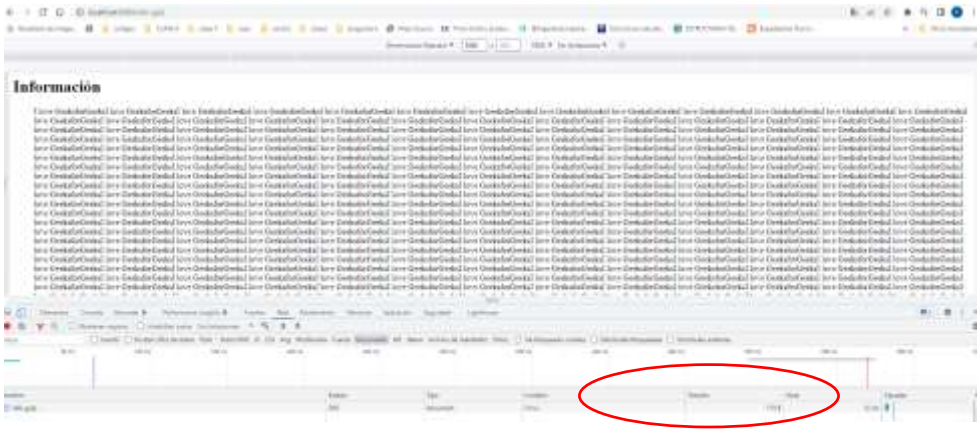
Como no se encontraban diferencias grandes en los resultados, y demostrar la eficiencia del uso de GZIP en la performance; se agregó una línea repetida ochocientas veces y donde se puede notar los beneficios de la compresión con GZIP.

```
//RUTA /info VISTA DATOS SENCILLA   CL28
route.get('/info', function(req, res) {
  let puerto = process.argv[3] || 8080
  const data = ('I love GeeksforGeeks').repeat(800) ;

  res.send(`<h1>Información</h1>
    <ul>
     ${data}
    <li>Argumentos de entrada: ${puerto} </li>
    <li>Nombre de la plataforma (sistema operativo): ${process.platform} </li>
    <li>Versión de node.js: ${process.version} </li>
    <li>Memoria total reservada (rss): ${process.memoryUsage.rss()} bytes </li>
    <li>Path de ejecución: ${process.execPath} </li>
    <li>Process id: ${process.pid}</li>
    <li>Carpeta del proyecto: ${process.cwd()} </li>
    <li>Número de procesadores en el servidor (workers): ${cpus.length} </li>
    </ul>`
  )
```

SIN GZIP - Tamaño 16.8kB



CON GZIP - Tamaño 713B



PERFORMANCE

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.

1- Realizamos el perfilamiento del servidor, realizando el test con --prof y analizamos los resultados obtenidos procesando con –prof-process

Podemos ver los comandos utilizados en consola:

node --prof src/server.js -p 8080 -m fork

artillery quick --count=20 --nun=50 http://localhost:8080/info > result_nobloq.txt

artillery quick --count=20 --num=50 http://localhost:8080/info-bloq > result_bloq.txt

node --prof-process bloq-v8.log > result_prof_bloq.txt

node --prof-process nobloq-v8.log > result_prof_nobloq.txt

Comparando los resultados obtenidos, observamos como el proceso no Bloqueante tarda la mitad del proceso bloqueante (con console.log(DATA)), siendo mucho más efectivo. Además del tiempo total, comparamos la media de respuesta y el tiempo medio de respuesta que necesita el usuario.

Comparación Files result_nobloq.txt y result_bloq.txt

	BLOQUEANTE	NO BLOQUEANTE
Total Time	10seg	5 seg
http.response_time:		
Media	47 s	10.9 s
vusers.session_length:		
Media	2369 mseg	658.6 mseg

Comparación de los resultados obtenidos del profile: result_prof_nobloq VS result_prof_bloq

	BLOQUEANTE	NO BLOQUEANTE
Ticks (Shared libraries)	5453	2439

Vemos que en Shared libraries el proceso no bloqueante se lleva muchos menos ticks (un poco menos de la mitad) de los que se lleva en el proceso bloqueante, siendo más eficiente.

Autocannon

Realizaremos test con Autocannon y obtendremos el diagram de flama con 0x. Para eso corremos los test en un archivo llamado benchmark.js donde llamamos a Autocanon.

Además, modificamos el package.json:

```
"scripts": {
  "test": "node ./src/benchmark.js",
  "start": "0x ./src/server.js"
},
```

Ahora prendemos el servidor con: npm start y en otra terminal ejecutamos npm test, corriendo ambos test en paralelo.

Podemos ver en las siguientes figuras el reporte generado por los test:

Los resultados para el proceso no bloqueante son mucho mejores, observando una mayor cantidad de respuestas por segundo (Req/sec) y una capacidad de procesamiento mejor. (Bytes/sec)

Resultados Proceso No Bloqueante

```
flore@HP-OMEN MINGW64 ~/Desktop/node/clase30---noooo (master)
$ npm test

> entrega-chat@1.0.0 test
> node ./src/benchmark.js

Running all benchmarks in parallel ...
Running 20s test @ http://localhost:8080/info
100 connections


┌─────────┬──────┬──────┬───────┬──────┬──────────┬─────────┬────────┐
│ Stat    │ 2.5% │ 50%  │ 97.5% │ 99%  │ Avg      │ Stdev   │ Max    │
├─────────┼──────┼──────┼───────┼──────┼──────────┼─────────┼────────┤
│ Latency │ 119 ms │ 185 ms │ 340 ms │ 384 ms │ 197.26 ms │ 56.57 ms │ 451 ms │
└─────────┴──────┴──────┴───────┴──────┴──────────┴─────────┴────────┘

┌───────────┬─────┬──────┬─────┬───────┬────────┬─────────┬────────┐
│ Stat      │ 1%  │ 2.5% │ 50% │ 97.5% │ Avg    │ Stdev   │ Min    │
├───────────┼─────┼──────┼─────┼───────┼────────┼─────────┼────────┤
│ Req/Sec   │ 404 │ 404  │ 511 │ 563   │ 503.6  │ 41.14   │ 404    │
├───────────┼─────┼──────┼─────┼───────┼────────┼─────────┼────────┤
│ Bytes/Sec │ 302 kB │ 302 kB │ 382 kB │ 421 kB │ 377 kB │ 30.8 kB │ 302 kB │
└───────────┴─────┴──────┴─────┴───────┴────────┴─────────┴────────┘

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

10k requests in 20.06s, 7.53 MB read
Running 20s test @ http://localhost:8080/info-bloq
100 connections
```

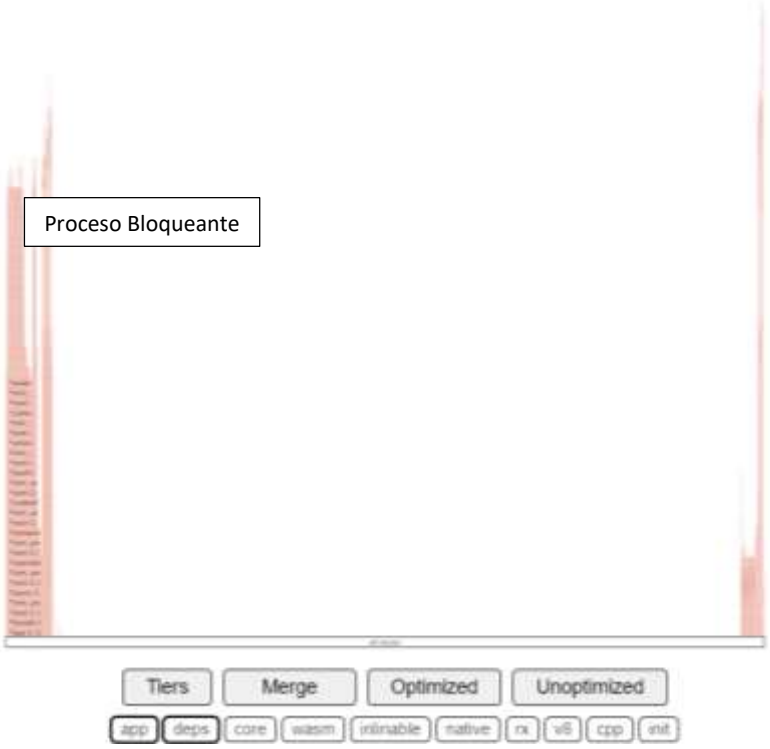
Resultados Proceso Bloqueante

```
┌─────────┬──────┬──────┬───────┬──────┬──────────┬─────────┬────────┐
│ Stat    │ 2.5% │ 50%  │ 97.5% │ 99%  │ Avg      │ Stdev   │ Max    │
├─────────┼──────┼──────┼───────┼──────┼──────────┼─────────┼────────┤
│ Latency │ 122 ms │ 186 ms │ 333 ms │ 369 ms │ 198.78 ms │ 56.3 ms │ 453 ms │
└─────────┴──────┴──────┴───────┴──────┴──────────┴─────────┴────────┘

┌───────────┬─────┬──────┬─────┬───────┬────────┬─────────┬────────┐
│ Stat      │ 1%  │ 2.5% │ 50% │ 97.5% │ Avg    │ Stdev   │ Min    │
├───────────┼─────┼──────┼─────┼───────┼────────┼─────────┼────────┤
│ Req/Sec   │ 356 │ 356  │ 508 │ 575   │ 501.2  │ 51.13   │ 356    │
├───────────┼─────┼──────┼─────┼───────┼────────┼─────────┼────────┤
│ Bytes/Sec │ 266 kB │ 266 kB │ 380 kB │ 430 kB │ 375 kB │ 38.3 kB │ 266 kB │
└───────────┴─────┴──────┴─────┴───────┴────────┴─────────┴────────┘

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

10k requests in 20.13s, 7.5 MB read
```

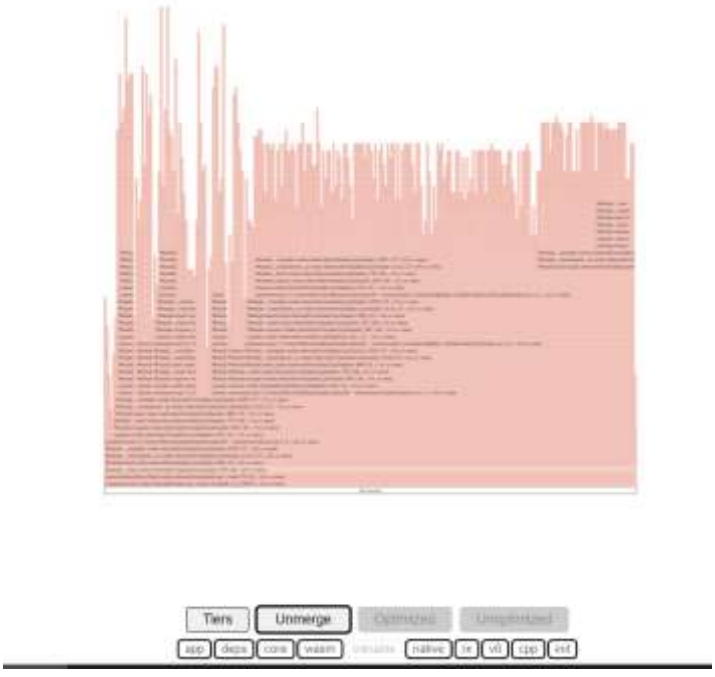
Mientras en la otra consola, termino de realizarse el diagrama de flama. Generandose una carpeta que contiene un archivo .html con el mismo.

```
Process exited, generating flamegraph
Flamegraph generated in
file://C:\Users\flore\Desktop\node\clase30---noooo\7476.0x\flamegraph.html
```

Del diagrama podemos observar ambos procesos:

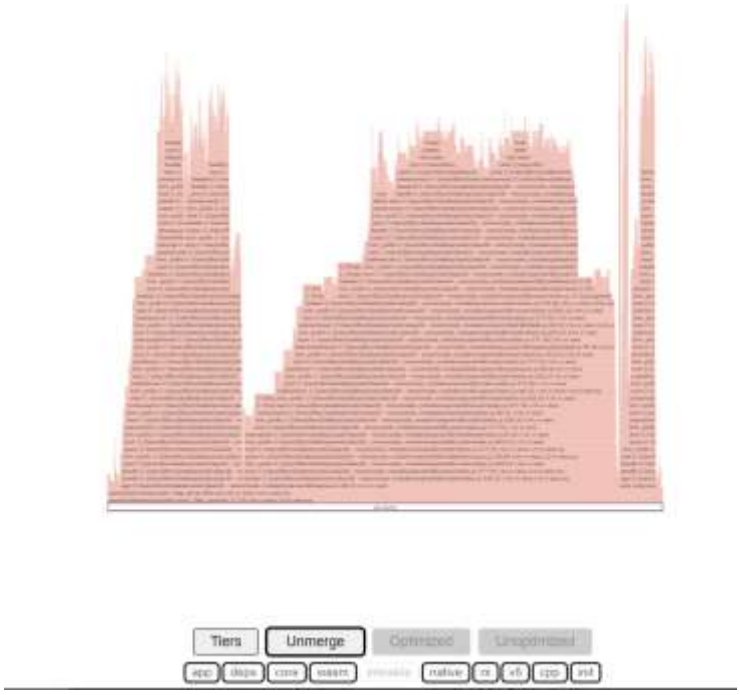


<u>Proceso Bloqueante</u>



Podemos observar como el proceso bloqueante tiene una mayor horizontalidad en su diagrama, lo que demuestra una mayor duración en el stack. Con procesos más largos.

Proceso No Bloqueante



Podemos observar como en el proceso no bloqueante tenemos una mayor cantidad de picos. Estos procesos que se encuentran más arriba y son más cortos tienen poca permanencia en el stack, por ende, son no bloqueantes.

La forma que debe tener el diagrama de flama en un proceso eficiente es con la mayor cantidad de picos y lo más fino posible.

Perfilamiento del servidor con el modo inspector de node.js

Revisar el tiempo de los procesos menos performantes sobre el archivo fuente de inspección.

node --inspect src/server.js

Proceso No Bloqueante

```
florw@HP-OMEN MINGW64 ~/Desktop/node/clase30---ncccc (master)
$ artillery quick --count=20 --num=50 http://localhost:8080/info
Running scenarios...
Phase started: unnamed (index: 0, duration: 1s) 15:21:02(-0300)

Phase completed: unnamed (index: 0, duration: 1s) 15:21:03(-0300)

All VUs finished. Total time: 10 seconds

--------------------------------
Summary report @ 15:21:10(-0300)
--------------------------------

http.codes.200: ..................................................................... 1000
http.request_rate: .................................................................. 239/sec
http.requests: ...................................................................... 1000
http.response_time:
  min: .............................................................................. 2
  max: .............................................................................. 64
  median: ........................................................................... 24.8
  p95: .............................................................................. 45.2
  p99: .............................................................................. 62.2
http.responses: ..................................................................... 1000
vusers.completed: ................................................................... 20
vusers.created: ..................................................................... 20
vusers.created_by_name.0: ........................................................... 20
vusers.failed: ...................................................................... 0
vusers.session_length:
  min: .............................................................................. 878.2
  max: .............................................................................. 1481.4
  median: ........................................................................... 1326.4
  p95: .............................................................................. 1465.9
  p99: .............................................................................. 1465.9
```

Proceso Bloqueante

```
florw@HP-OMEN MINGW64 ~/Desktop/node/clase30---ncccc (master)
$ artillery quick --count=20 --num=50 http://localhost:8080/info-bloq
Running scenarios...
Phase started: unnamed (index: 0, duration: 1s) 15:37:28(-0300)

Phase completed: unnamed (index: 0, duration: 1s) 15:37:21(-0300)

All VUs finished. Total time: 10 seconds

--------------------------------
Summary report @ 15:37:28(-0300)
--------------------------------

http.codes.200: ..................................................................... 1000
http.request_rate: .................................................................. 83/sec
http.requests: ...................................................................... 1000
http.response_time:
  min: .............................................................................. 12
  max: .............................................................................. 182
  median: ........................................................................... 102.5
  p95: .............................................................................. 144
  p99: .............................................................................. 172.5
http.responses: ..................................................................... 1000
vusers.completed: ................................................................... 20
vusers.created: ..................................................................... 20
vusers.created_by_name.0: ........................................................... 20
vusers.failed: ...................................................................... 0
vusers.session_length:
  min: .............................................................................. 4449.8
  max: .............................................................................. 5414.2
  median: ........................................................................... 5272.4
  p95: .............................................................................. 5378.9
  p99: .............................................................................. 5378.9
```

Resultados Chrome://inspect

Podemos ver los tiempos que necesita el proceso

```
const { Router } =require( 'express');
const route= Router();
const path = require( 'path')
const {faker}  =require( "@faker-js/faker");
faker.locale='es'
// const express= require ("express")
// const compression = require ("compression")
// const app = express()


const cpus = require ('os').cpus()
const logger = require('../utils/logger.js')
//middleware a nivel enrutador, se ejecutará en cada req del route
route.use(function (req, res, next){
    const { url, method } = req;
    logger.info(`Método ${method} URL ${url} recibida`);
    next()
})

module.export = route.get ('/api/productos-test', (req,res)=>{

    const response = [];

    for (let i = 0; i < 5; i++) {
```

Ahora en nuestra ruta: /info

```
index.js  index.js ×
const { Router } =require( 'express');
const route= Router();
const path = require( 'path')
const {faker}  =require( "@faker-js/faker");
faker.locale='es'
// const express= require ("express")
// const compression = require ("compression")
// const app = express()


const cpus = require ('os').cpus()
const logger = require('../utils/logger.js')
//middleware a nivel enrutador, se ejecutará en cada req del route
route.use(function (req, res, next){
    const { url, method } = req;
    logger.info(`Método ${method} URL ${url} recibida`);
    next()
})

module.export = route.get ('/api/productos-test', (req,res)=>{

    const response = [];

    for (let i = 0; i < 5; i++) {
      response.push({
        title: faker.commerce.product(),
```

Ahora en nuestra ruta: /info-bloq

```
    }
  })
})

//RUTA /info VISTA DATOS SENCILLA   CL28
route.get('/info', function(req, res) {
  let puerto = process.argv[3] || 8080
  const data = `<h1>Información</h1>
  <ul>
  <li>Argumentos de entrada: ${puerto} </li>
  <li>Nombre de la plataforma (sistema operativo): ${process.platform} </li>
  <li>Versión de node.js: ${process.version} </li>
  <li>Memoria total reservada (rss): ${process.memoryUsage.rss()} bytes </li>
  <li>Path de ejecución: ${process.execPath} </li>
  <li>Process id: ${process.pid}</li>
  <li>Carpeta del proyecto: ${process.cwd()} </li>
  <li>Número de procesadores en el servidor (workers): ${cpus.length} </li>
  </ul>` ;

  res.send(data)

})
route.get('/info-bloq', function(req, res) {
```

```
  res.send(data)

})
route.get('/info-bloq', function(req, res) {
  let puerto = process.argv[3] || 8080
  const data = `<h1>Información</h1>
  <ul>
  <li>Argumentos de entrada: ${puerto} </li>
  <li>Nombre de la plataforma (sistema operativo): ${process.platform} </li>
  <li>Versión de node.js: ${process.version} </li>
  <li>Memoria total reservada (rss): ${process.memoryUsage.rss()} bytes </li>
  <li>Path de ejecución: ${process.execPath} </li>
  <li>Process id: ${process.pid}</li>
  <li>Carpeta del proyecto: ${process.cwd()} </li>
  <li>Número de procesadores en el servidor (workers): ${cpus.length} </li>
  </ul>` ;
  console.log(data)
  res.send(data)

})
```

Ahora generaremos un cuadro comparativo con los diferentes resultados obtenidos:

	No Bloqueante (/info)	Bloqueante (/info-bloq)
http.response_time: (Media)	24.8	102.5
vusers.session_length: (Media)	1326.4	5272.4
Tiempo del proceso en Chrome/inspect (archivo: index.js)	17.9 ms	31.4 ms

CONCLUSIÓN:

Podemos ver como se optimizan los procesos y el rendimiento, gracias a la compresión de nuestra aplicación con GZIP.

Además, podemos ver como los procesos no bloqueantes son mucho más efectivos mejorando notoriamente los tiempos de respuesta de nuestra aplicación.