CODER HOUSE

Clase 32. Programación Backend

Logs, profiling & debug Parte II

OBJETIVOS DE LA CLASE

- Utilizar Artillery para realizar test de carga a servidores manuales.
- Realizar Profiling con Node built-in Profiler.
- Explorar y utilizar Autocannon para realizar test de carga y 0x para generar y analizar gráficos de flama interactivos.

CODER HOUSE

CRONOGRAMA DEL CURSO

Clase 31

Logs, profiling & debug Parte I

Clase 32

Logs, profiling & debug Parte II

Clase 33

Productos Cloud: Heroku

CODER HOUSE

ARTILLERY

¿Qué es?

- Artillery es una dependencia de Node moderna, potente, fácil y muy útil para realizar test de carga a servidores.
- Cuenta con un conjunto de herramientas para tests de performance que se usa para enviar aplicaciones escalables que se mantengan eficaces y resistentes bajo cargas elevadas.
- Podemos usar Artillery para ejecutar dos tipos de pruebas de rendimiento:
    - Pruebas que cargan un sistema, o sea, **pruebas de carga**, de estrés.
    - Pruebas que verifican que un sistema funciona como se esperaba, es decir, **pruebas funcionales continuas**.

👉 Solo puede ser utilizado en sistemas de backend, no se puede utilizar en el front.

Usando Artillery

Ejemplo en vivo

1. Para empezar a usar Artillery, primero vamos a instalarlo.

```
$ npm install -g artillery
```

2. A continuación, creamos un servidor. En este caso, vamos a poder encender el servidor en modo Fork o en modo Cluster.

```
import express from 'express'
import cluster from 'cluster'
import { cpus } from 'os'

const PORT = parseInt(process.argv[2]) || 8080
const modoCluster = process.argv[3] == 'CLUSTER'
```

Ejemplo en vivo

3. De esta forma terminamos de configurar el servidor:

```
if (modoCluster && cluster.isPrimary) {
    const numCPUs = cpus().length

    console.log(`Número de procesadores:${numCPUs}`)
    console.log(`PID MASTER ${process.pid}`)

    for (let i = 0; i < numCPUs; i++) {
        cluster.fork()
    }

    cluster.on('exit', worker => {
        console.log('Worker', worker.process.pid, 'died', new Date().toLocaleString())
        cluster.fork()
    })
} else {
    const app = express()

    app.get('/', (req, res) => {
        const primes = []
        const max = Number(req.query.max) || 1000
        for (let i = 1; i <= max; i++) {
            if (isPrime(i)) primes.push(i)
        }
        res.json(primes)
    })

    app.listen(PORT, () => {
        console.log(`Servidor express escuchando en el puerto ${PORT}`)
        console.log(`PID WORKER ${process.pid}`)
    })
}
```

CODER HOUSE

Usando Artillery: Función isPrime

Ejemplo en vivo

👇 Esta es la función isPrime que utilizamos en el servidor.

```
function isPrime(num) {
   if ([2, 3].includes(num)) return true;
   else if ([2, 3].some(n => num % n == 0)) return false;
   else {
       let i = 5, w = 2;
       while ((i ** 2) <= num) {
           if (num % i == 0) return false
           i += w
           w = 6 - w
       }
   }
   return true
}
```

➔ Su única funciones es recibir un número como parámetro y retornar *true* si el número es primo o *false* si no lo es.

➔ Lo que vamos a hacer es realizar el test de carga sobre esta función desde la ruta *get* “/” configurada en el servidor.

Usando Artillery

Ejemplo en vivo

1. Prendemos el servidor en modo Fork con el comando: `$ node server.js 8081 FORK`

2. Abrimos otra terminal sobre la carpeta del proyecto y con el siguiente comando hacemos el test de carga:

```
artillery quick --count 50 -n 40 http://localhost:8081?max=100000 > result_fork.txt
```

El resultado se va a guardar en el archivo *result_fork.txt*. Para poder visualizar el archivo, debemos apagar el servidor una vez finalizado el test.

3. Hacemos lo mismo con el servidor en modo Cluster: `$ node server.js 8081 CLUSTER`

```
artillery quick --count 50 -n 40 http://localhost:8081?max=100000 > result_cluster.txt
```

📝 **NOTA:** En las *query* seteamos *max en 100000* para calcular números primos hasta 100.000.

CODER HOUSE

Comparando los resultados

Podemos ahora comparar los resultados de ambos archivos. Nos interesará fijamos en la última parte de los mismos:

result_fork.txt

```
All virtual users finished
Summary report @ 19:37:05(-0300) 2021-05-04
  Scenarios launched:  50
  Scenarios completed: 50
  Requests completed:  2000
  Mean response/sec: 361.66
  Response time (msec):
    min: 3
    max: 179
    median: 70
    p95: 127
    p99: 157
  Scenario counts:
    0: 50 (100%)
  Codes:
    404: 2000
```

result_cluster.txt

```
All virtual users finished
Summary report @ 19:37:32(-0300) 2021-05-04
  Scenarios launched:  50
  Scenarios completed: 50
  Requests completed:  2000
  Mean response/sec: 493.83
  Response time (msec):
    min: 0
    max: 139
    median: 49
    p95: 83
    p99: 103.5
  Scenario counts:
    0: 50 (100%)
  Codes:
    404: 2000
```

CODER HOUSE

Comparando los resultados

result_fork.txt

```
All virtual users finished
Summary report @ 19:37:05(-0300) 2021-05-04
  Scenarios launched:  50
  Scenarios completed: 50
  Requests completed:  2000
  Mean response/sec: 361.66
  Response time (msec):
    min: 3
    max: 179
    median: 70
    p95: 127
    p99: 157
  Scenario counts:
    0: 50 (100%)
  Codes:
    404: 2000
```

result_cluster.txt

```
All virtual users finished
Summary report @ 19:37:32(-0300) 2021-05-04
  Scenarios launched:  50
  Scenarios completed: 50
  Requests completed:  2000
  Mean response/sec: 493.83
  Response time (msec):
    min: 0
    max: 139
    median: 49
    p95: 83
    p99: 103.5
  Scenario counts:
    0: 50 (100%)
  Codes:
    404: 2000
```

¿Qué vemos?

- Podemos ver que la media de respuestas (Mean response/sec) por segundo es mucho más alta en el Cluster, por lo que, comprobamos que es más eficiente.
- Los milisegundos de latencia (Response time, median) es más alto en el Fork que en Cluster. Por lo que se vuelve a comprobar que es mejor el servidor en modo Cluster.

CODER HOUSE

PROFILING

¿Qué es?

- ***Profiling*** en español es ***análisis de rendimiento***. Es la investigación del comportamiento de un programa usando información reunida desde el análisis dinámico del mismo.
- El objetivo es averiguar el tiempo dedicado a la ejecución de diferentes partes del programa para detectar los puntos problemáticos y las áreas donde sea posible llevar a cabo una optimización del rendimiento (ya sea en velocidad o en consumo de recursos).
- Un profiler puede proporcionar distintas salidas, como una traza de ejecución o un resumen estadístico de los eventos observados.

Profiling en NodeJs

- Cualquier navegador moderno, como Google Chrome, tiene un built-in profiler integrado en DevTools, que registra toda la información sobre las funciones y cuánto tiempo lleva ejecutarlas en un archivo de registro.
- Luego, el navegador analiza este archivo de *log*, brindándonos información legible sobre lo que está sucediendo, para que podamos entenderlo y encontrar cuellos de botella.
- Node también tiene un built-in profiler pero con una diferencia: este no analiza archivos de *log* como los navegadores, sino que simplemente recopila toda la información en estos archivos de *log*.
- Significa que necesita tener alguna herramienta separada que pueda comprender este archivo de *log* y proporcionarnos la información de forma legible.

Antes de empezar a ver cómo utilizamos el Node *built-in profiler*, vamos a ver qué es Curl y cómo lo podemos usar para esto.

CODER HOUSE

Curl

- ***Curl*** es una herramienta de línea de comandos y librería para transferir datos con URL. Se usa en líneas de comando o scripts para transferir datos.
- Es utilizado a diario por prácticamente todos los usuarios de Internet en el mundo.
- Además, se utiliza en automóviles, televisores, teléfonos móviles, tabletas, entre otros y es el motor de transferencia de Internet para miles de aplicaciones de software en más de diez mil millones de instalaciones.

Curl - instalación

1. Para usarlo, debemos descargarlo e instalarlo. Lo podemos hacer desde: https://curl.se/download.html
2. Una vez descargado, descomprimimos el zip y en la carpeta “bin” encontramos el archivo de instalación “*.exe”* llamado *curl.exe*.
   (Posiblemente tengamos que ejecutarlo como administrador)
3. Una vez instalado, ya lo podemos utilizar como comando en la consola.

En la explicación que sigue vamos a estar usándolo.

Ejemplo en vivo

Node built-in profiler

- La mayoría de las veces, es más fácil usar el profiler que ya tiene Node, en lugar de usar otra herramienta para esto.
- Para empezar a usar este profiler, primero creamos una pequeña aplicación en Express con un servidor y algunas rutas.

Configuramos el archivo server.js como sigue:

```
const express = require("express");
const crypto = require("crypto");

const app = express();

const users = {}

app.use(express.static('public'))
```

```
const PORT = parseInt(process.argv[2]) || 8080;
const server = app.listen(PORT, () => {
  console.log(`Servidor escuchando en el puerto ${PORT}`);
});
server.on("error", (error) => console.log(`Error en servidor: ${error}`));
```

Node built-in profiler

Ejemplo en vivo

```
app.get("/getUsers", (req, res) => {
  res.json({ users })
})

app.get("/newUser", (req, res) => {
  let username = req.query.username || "";
  const password = req.query.password || "";

  username = username.replace(/[!@#$%^&*]/g, "");

  if (!username || !password || users[username]) {
    return res.sendStatus(400);
  }

  const salt = crypto.randomBytes(128).toString("base64");
  const hash = crypto.pbkdf2Sync(password, salt, 10000, 512, "sha512");

  users[username] = { salt, hash };

  res.sendStatus(200);
});
```

- Con la ruta */getUsers* se muestra el listado de usuarios registrados.
- Con la ruta */newUser* se registra un nuevo usuario.
- Se utiliza el módulo *crypto* para encriptar las contraseñas.

CODER HOUSE

Node built-in profiler: auth-bloq

Ejemplo en vivo

```
app.get("/auth-bloq", (req, res) => {
  let username = req.query.username || "";
  const password = req.query.password || "";

  username = username.replace(/[!@#$%^&*]/g, "");

  if (!username || !password || !users[username]) {
    process.exit(1)
    // return res.sendStatus(400);
  }

  const { salt, hash } = users[username];
  const encryptHash = crypto.pbkdf2Sync(password, salt, 10000, 512, "sha512");

  if (crypto.timingSafeEqual(hash, encryptHash)) {
    res.sendStatus(200);
  } else {
    process.exit(1)
    // res.sendStatus(401);
  }
});
```

La ruta */auth-bloq* realiza el login del usuario.

En este caso, el proceso por el cual se realizar el login es sincrónico, por lo tanto es un proceso bloqueante.

CODER HOUSE

Node built-in profiler: auth-nobloq

Ejemplo en vivo

- La ruta */auth-nobloq* también realiza el login del usuario.
- En este caso, el proceso por el cual se realizar el login es asincrónico, por lo tanto es un proceso **NO** bloqueante.

```
app.get("/auth-nobloq", (req, res) => {
  let username = req.query.username || "";
  const password = req.query.password || "";
  username = username.replace(/[!@#$%^&*]/g, "");

  if (!username || !password || !users[username]) {
    process.exit(1)
    // return res.sendStatus(400);
  }
  crypto.pbkdf2(password, users[username].salt, 10000, 512, 'sha512', (err, hash) => {
    if (users[username].hash.toString() === hash.toString()) {
      res.sendStatus(200);
    } else {
      process.exit(1)
      //res.sendStatus(401);
    }
  });
});
```

CODER HOUSE

Ejemplo en vivo

Node built-in profiler

- Una vez configurado el servidor y las rutas, vamos a usar nuevamente Artillery para realizar los test de carga y obtener la información necesaria.

- Finalmente, ya estamos listos para poder prender el servidor, y lo vamos a hacer en modo profiler.

  Para eso usamos el siguiente comando: `$ node --prof server.js`

Node built-in profiler

Ejemplo en vivo

Ahora abrimos otra terminal, en la carpeta del proyecto, y procedemos con los siguientes pasos.

1. Con el siguiente comando, creamos un nuevo usuario:

```
curl -X GET "http://localhost:8080/newUser?username=marian&password=qwerty123"
```

2. Ahora, puedo usar el test de carga. Para eso, utilizo el siguiente comando:

```
artillery quick --count 10 -n 50 "http://localhost:8080/auth-bloq?username=marian&password=qwerty123" > result_bloq.txt
```

Va a hacer un test de 10 *request* con 50 usuarios a la url especificada. Y el resultado lo va a guardar en el archivo *result_bloq.txt*. Para ver este archivo debo salir del servidor (sino no nos deja abrirlo).

Node built-in profiler

Ejemplo en vivo

Con lo hecho, se crea también un archivo llamado *Isolate* que está encriptado. Primero, lo debemos renombramos como *bloq-v8.log* y antes de decodificar el archivo, hacemos lo mismo pero cuando la ruta es la no bloqueante.

1. Creamos nuevamente un nuevo usuario igual que antes.

```
curl -X GET "http://localhost:8080/newUser?username=dani&password=qwerty123"
```

2. Luego, el comando es similar pero la url es hacia la ruta no bloqueante de login

```
artillery quick --count 10 -n 50 "http://localhost:8080/auth-nobloq?username=dani&password=qwerty123" > result_nobloq.txt
```

Ejemplo en vivo

Node built-in profiler

Pasamos ahora a decodificar los archivos log que se crearon.

- Para eso, utilizamos el siguiente comando para cada uno:

```
node --prof-process bloq-v8.log > result_prof-bloq.txt

node --prof-process nobloq-v8.log > result_prof-nobloq.txt
```

- Se crean entonces los archivos *result_prof-bloq.txt* y *result_prof-nobloq.txt* con la información de los primeros archivos decodificada.

Node built-in profiler: Comparación

Veamos una comparación entre los ticks de cada uno de los procesos.

result_prof-bloq.txt

```
[Summary]:
  ticks  total  nonlib   name
      1    0.0%  100.0%  JavaScript
      0    0.0%    0.0%  C++
      4    0.1%  400.0%  GC
   6746  100.0%          Shared libraries
```

result_prof-nobloq.txt

```
[Summary]:
  ticks  total  nonlib   name
      2    0.1%  100.0%  JavaScript
      0    0.0%    0.0%  C++
      6    0.2%  300.0%  GC
   3205   99.9%          Shared libraries
```

CODER HOUSE

Node built-in profiler: Comparación

result_prof-bloq.txt

```
[Summary]:
  ticks  total  nonlib   name
      1   0.0%  100.0%  JavaScript
      0   0.0%    0.0%  C++
      4   0.1%  400.0%  GC
   6746  100.0%          Shared libraries
```

result_prof-nobloq.txt

```
[Summary]:
  ticks  total  nonlib   name
      2   0.1%  100.0%  JavaScript
      0   0.0%    0.0%  C++
      6   0.2%  300.0%  GC
   3205  99.9%          Shared libraries
```

¿Qué vemos?

- 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.
- De esta forma, y analizando ambos archivos por completo, podemos ir haciendo una inspección de los procesos.

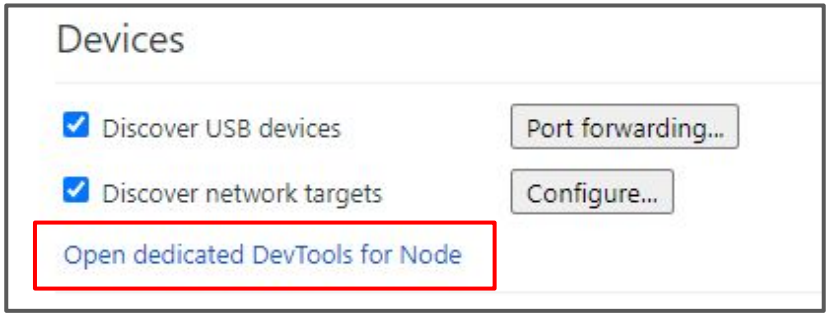
Resulta una tarea bastante manual de esta forma. Veamos qué podemos hacer...

Node inspect

Ejemplo en vivo

Veamos otro modo de hacer lo recién visto con pasos más sencillos.

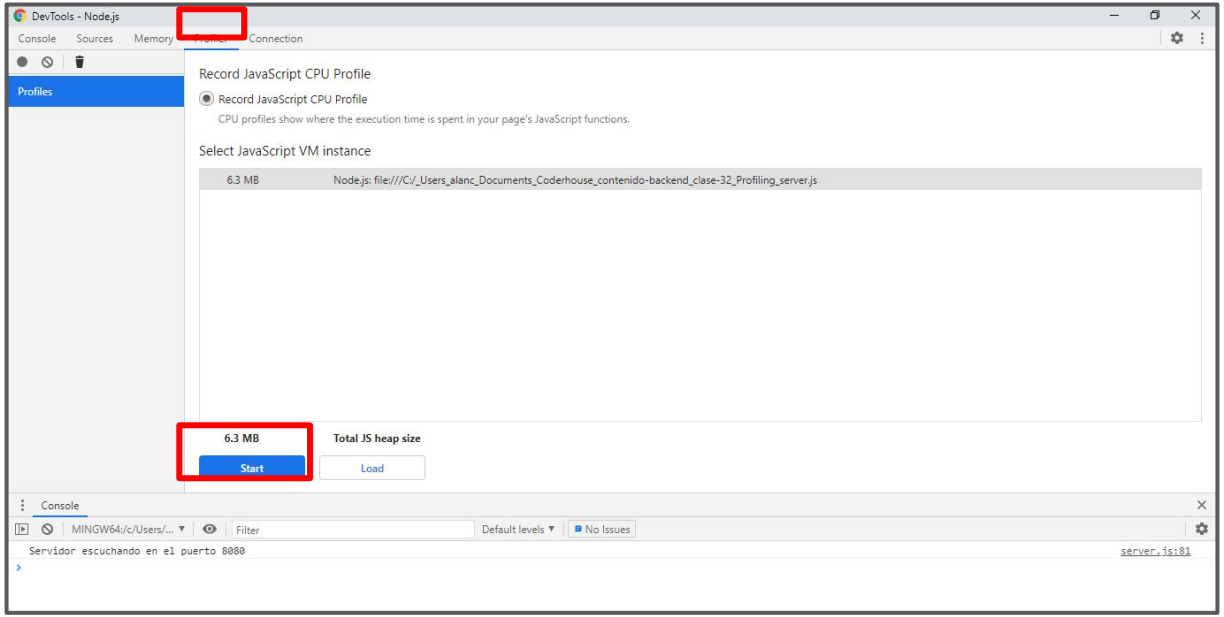
1. Para eso, primero prendemos el servidor con el comando: `node --inspect server.js`
2. Luego, en el navegador Google Chrome, ponemos: `chrome://inspect`
3. Abrimos el DevTools:



Node inspect

Ejemplo en vivo

4. Se nos abre una nueva ventana, y vamos a la pestaña *profiler*. Allí, cliqueo en el botón de *start*.



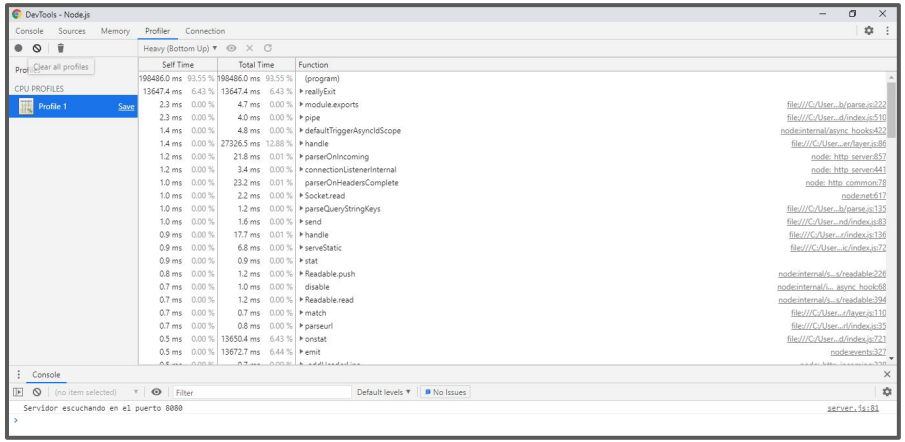
CODER HOUSE

Node inspect

Ejemplo en vivo

5. Una vez hecho esto, puedo volver a la consola y correr nuevamente los comandos del test de carga artillery que mencionamos anteriormente. Para los procesos bloqueante y no bloqueante.

6. Una vez que finaliza, ponemos el botón stop en el navegador, y nos muestra algo como esto, con la misma información que los archivos vistos anteriormente.

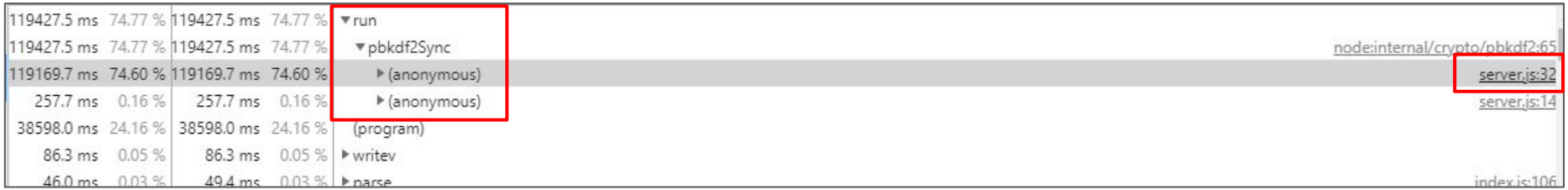


CODER HOUSE

Node inspect

Ejemplo en vivo

- Si vamos a “run” y luego desplegamos también el proceso que se nos abre, podemos ver, mirando a la derecha que en el archivo server.js línea 32 tenemos un proceso bloqueantes.
- Podemos entonces hacer click sobre eso (*server.js:32*).



CODER HOUSE

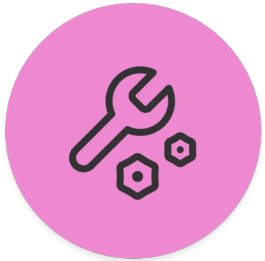
Node inspect

Ejemplo en vivo

```
11          res.json({users})
12      })
13
14      app.get("/newUser", (req, res) => {
15          let username = req.query.username || "";
16          const password = req.query.password || "";
17
18  0.1 ms  username = username.replace(/[!@#$%^&*]/g, "");
19
20          if (!username || !password || users[username]) {
21              return res.sendStatus(400);
22          }
23
24  0.3 ms  const salt = crypto.randomBytes(128).toString("base64");
25  0.3 ms  const hash = crypto.pbkdf2Sync(password, salt, 10000, 512, "sha512");
26
27  0.1 ms  users[username] = { salt, hash };
28
29  0.1 ms  res.sendStatus(200);
30      });
31
32      app.get("/auth-blog", (req, res) => {
33  1.7 ms  let username = req.query.username || "";
34  0.1 ms  const password = req.query.password || "";
35
36  2.3 ms  username = username.replace(/[!@#$%^&*]/g, "");
37
38  0.4 ms  if (!username || !password || !users[username]) {
39              process.exit(1)
40              //return res.sendStatus(400);
41          }
42
```

- Con esto, se nos abre la vista del código, junto con el tiempo en milisegundos que demoró esa función.
- Entonces podemos ver los milisegundos de cada función y ver las que están demorando la ejecución de la aplicación.

CODER HOUSE



ANÁLISIS DE PERFORMANCE: bloq vs no-bloq

Tiempo: 10 minutos

CODER HOUSE

Análisis de performance con Profiler

Desafío generico

Tiempo: 10 minutos

1. Realizar un servidor que calcule 10000 números aleatorios entre el 0 y el 9 inclusive.
    - El servidor devolverá los números calculados en un array dentro de un objeto en formato JSON: {randoms: [array de randoms]}.
    - Se van a utilizar dos rutas en las que el cliente puede requerir la información: '/randoms-nodebug' y '/randoms-debug', la última contendrá un console.log que enviará el array calculado a la consola del servidor.

Análisis de performance con Profiler

Desafío generico

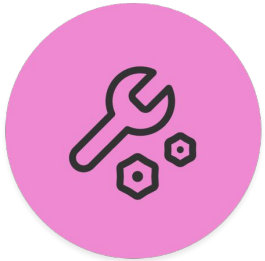
Tiempo: 10 minutos

2. Realizar un análisis de performance a través del profiler (--prof) de node.js y del modo inspect (--inspect) para las dos rutas, utilizando Artillery como generador de carga con 50 usuarios virtuales emitiendo 50 request cada uno.

    - Verificar que los tiempos de proceso en la ruta '/randoms-debug' sean mayores a los de la ruta '/randoms-nodebug', debido a la operación sincrónica bloqueante del console.log.

BREAK

¡5/10 MINUTOS Y VOLVEMOS!



ANÁLISIS DE PERFORMANCE: fork vs cluster

Tiempo: 10 minutos

CODER HOUSE

Análisis de performance con Artillery

Desafío generico

Tiempo: 10 minutos

Utilizando el desafío anterior, agregar un parámetro (FORK ó CLUSTER) en línea de comandos que permita habilitar o deshabilitar el modo cluster en el servidor.

- Realizar un análisis de performance sobre la ruta '/randoms-debug' en modo fork y cluster utilizando Artillery como generador de carga con 50 usuarios virtuales con 50 request cada uno.
- Revisar los reportes de Artillery, corroborando que los request por segundo y la latencia del servidor en modo cluster tengan mejores resultados que en modo fork.

AUTOCANNON Y 0x

CODER HOUSE

¿Qué son?

- **Autocannon** es una dependencia de Node (similar a Artillery) que nos ayuda a realizar los test de carga.
- Es una herramienta de evaluación comparativa HTTP / 1.1.

¿Qué son?

- **0x** es una dependencia que perfila y genera un gráfico de flama (*flame graph*) interactivo para un proceso Node en un solo comando.
- En este caso, vamos a hacer los test de carga por código, en lugar de por consola como hicimos con Artillery.

Ejemplo en vivo

Realizando test de carga

1. Para empezar debemos como siempre crear un servidor. Vamos a usar exactamente el mismo que usamos en el ejemplo de Profiling.
2. Generamos la ruta de registro de usuario (/newUser) y las rutas de login bloqueante (/auth-bloq) y no bloqueante (/auth-nobloq).
3. Procedemos a instalar Autocannon y 0x:
    a. Autocannon lo vamos a instalar como dependencia del proyecto, ya que como dijimos, los test los vamos a realizar por código.
    b. 0x lo instalamos de forma global.

```
$ npm install autocannon --save
```

```
$ npm install -g  0x
```

CODER HOUSE

Realizando test de carga

Ejemplo en vivo

```
const autocannon = require('autocannon')
const { PassThrough } = require('stream')

function run (url) {
  const buf = []
  const outputStream = new PassThrough()

  const inst = autocannon({
    url,
    connections: 100,
    duration: 20
  })

  autocannon.track(inst, { outputStream })

  outputStream.on('data', data => buf.push(data))
  inst.on('done', function () {
    process.stdout.write(Buffer.concat(buf))
  })
}

console.log('Running all benchmarks in parallel ...')

run('http://localhost:8080/auth-bloq?username=dani&password=qwerty123')
run('http://localhost:8080/auth-nobloq?username=dani&password=qwerty123')
```

4. Hacemos parecido a lo anterior, y con la función ***run*** ejecutamos el test para la ruta del proceso bloqueante y para la del no bloqueante.

- ➔ Los test los vamos a realizar en un archivo llamado *benchmark.js*.
- ➔ En él, requerimos Autocannon y creamos el test.

CODER HOUSE

Ejemplo en vivo

Realizando test de carga

En el *package.json* cambiamos en el *script "start"*, en vez de Node como siempre, ponemos 0x.

- Con esto, lo que hacemos es que se genere el gráfico de flama.
- Además, en el *script "test"* tenemos que decirle qué archivo va a testear, aclarando que es de Node (como vemos en el código).

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

Realizando test de carga: ¡ahora sí!

Ejemplo en vivo

- Prendemos el servidor con `$ npm start`
- Luego, en otra terminal, registramos un usuario, con el comando que usamos en profiling.

```
$ curl -X GET "http://localhost:8080/newUser?username=dani&password=qwerty123"
```

- Finalmente, ejecutamos los test con el comando `$ npm test`
- Ambos test se ejecutan en paralelo.
- Se genera en consola un reporte parecido a los que vimos con los métodos anteriores. Uno por cada test.

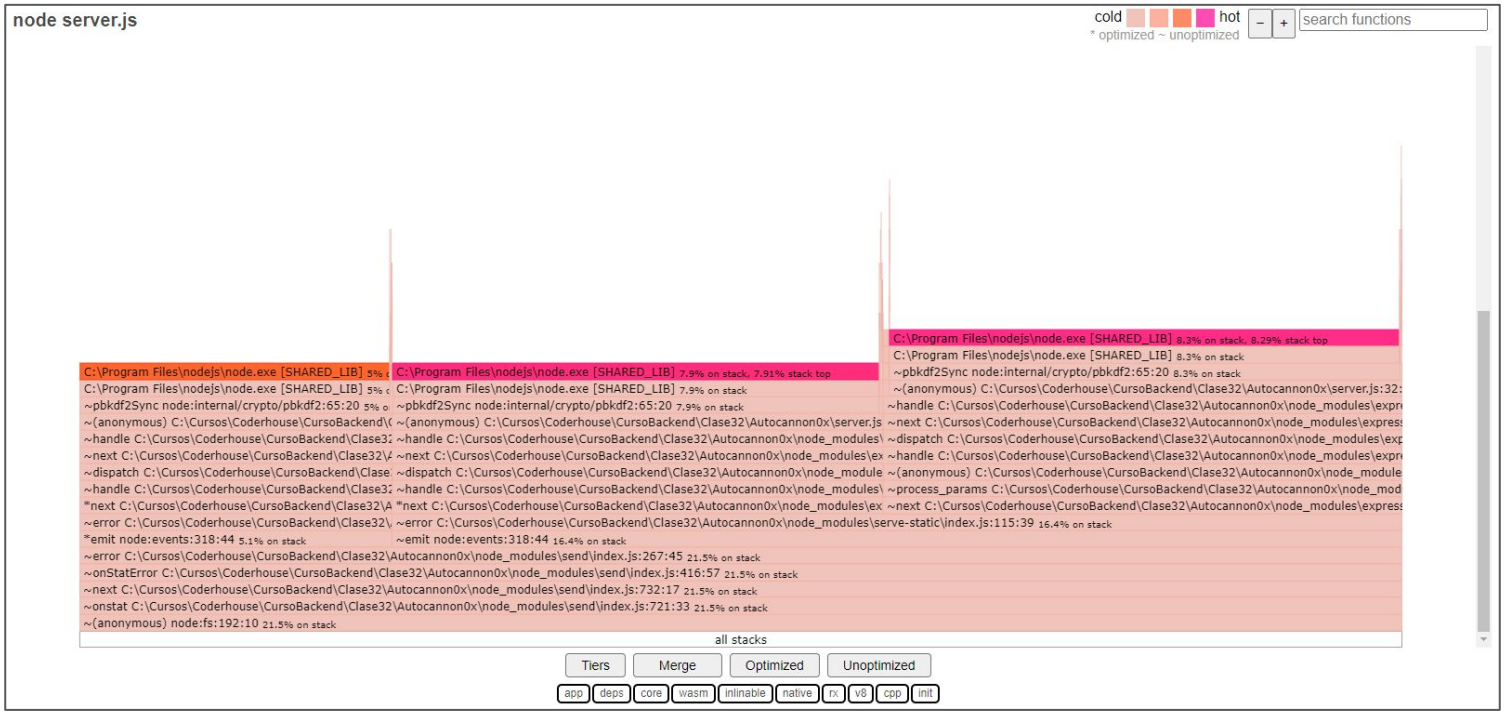
Realizando test de carga: diagramas HTML

Ejemplo en vivo

- Cuando apagamos el servidor, se crea una carpeta de nombre aleatorio.
- Esta contiene los resultados en archivo Isolate, similar a los visto y además un html con los diagramas de flama.
- Este archivo HTML lo podemos abrir en un navegador, y de esa forma podemos ver los diagramas.

Ejemplo diagrama de proceso bloqueante

Este tipo de diagrama veremos en el caso de un proceso bloqueante:



CODER HOUSE

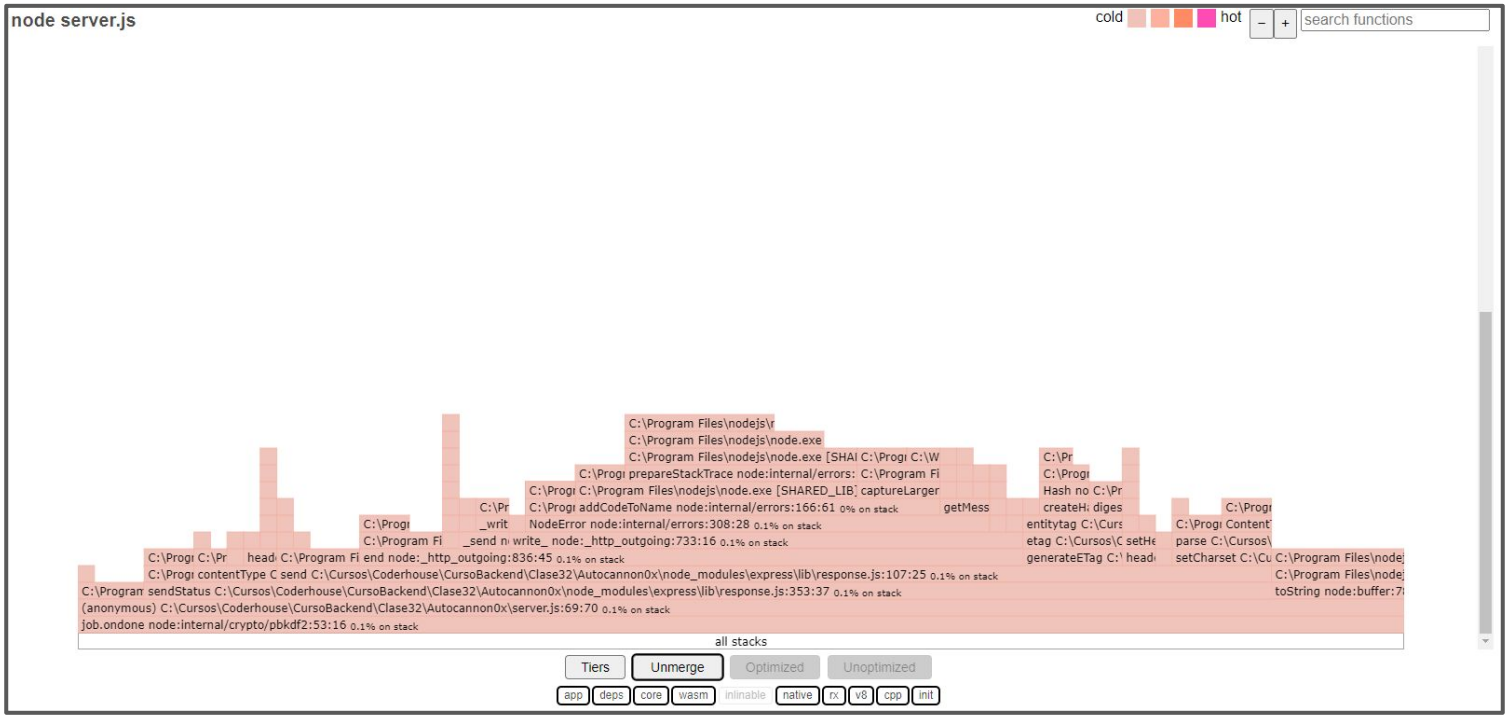
Explorar y navegar el diagrama

Ejemplo en vivo

- Con los botones de abajo en el medio, podemos ir cambiando el color, eligiendo qué procesos se muestran, cuáles no, etc.
- La altura del diagrama representa la profundidad del stack de Node. Cuanto más arriba llegue el diagrama de flamas, más anidado está dentro del stack de procesos.
- Los procesos de más arriba son los que primero tienen que resolverse para dejar liberados los procesos de abajo. Es decir, los procesos de arriba son los que bloquean a los de abajo y son los que están en color más oscuro (“hot”).
- Esto es justamente porque es el test sobre el proceso bloqueante.
- La horizontalidad es la duración en el stack, entonces cuanto más largos sean, mayor duración tienen. Por eso observamos que en el proceso bloqueante, los procesos duran mucho tiempo en el stack. Y con esto vemos también su planitud, no tiene picos, como si tiene el no bloqueante.

Ejemplo diagrama de proceso no bloqueante

En el caso del proceso no bloqueante el diagrama va a tener esta forma:



👉 Vemos que los procesos tienen todos el mismo color, propio del tipo no bloqueante.

CODER HOUSE

Análisis diagrama de proceso no bloqueante

- Al ver la horizontalidad, vemos que los varios procesos, sobretodo los que están más arriba, son cortos, y esa es su duración en el stack, por lo que al durar poco no bloquean a los que siguen más abajo, a diferencia de lo que pasaba en el proceso bloqueante.
- Entonces, estos procesos cortos tienen poca permanencia en el stack y son por ende, no bloqueantes. Se observan picos que no existían en el proceso bloqueante.
- La forma que deberían tener los diagramas de flama para que el proceso sea eficiente es con la mayor cantidad de picos y lo más finos posibles.

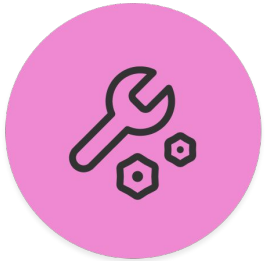
Bloqueante vs No Bloqueante

- Para poder compararlos mejor, a la izquierda vemos el proceso bloqueante y a la derecha el no bloqueante, ambos con la misma escala.
- El no bloqueante casi no lo vemos, por lo que podemos decir que sus procesos duran muchísimo menos tiempo en el stack (mi finos) como debe suceder en procesos más eficientes.

Bloqueante

No Bloqueante

CODER HOUSE



ANÁLISIS DE PERFORMANCE CON AUTOCANNON Y 0x

Tiempo: 10 minutos

CODER HOUSE

Análisis de performance con Autocannon y 0x

Desafío generico

Tiempo: 10 minutos

1. Realizar un análisis de performance sobre el desafío anterior, utilizando 0x y autocannon en modo consola. Con autocannon realizar un test con el servidor corriendo con 0x, emulando 500 conexiones concurrentes realizadas en 20 segundos de test.

2. Hacer el procedimiento mencionado con el servidor en modo fork y sobre los endpoint '/randoms-debug' y '/randoms-nodebug' obteniendo en cada caso el reporte de autocannon y el diagrama de flama.

3. Analizar para cada caso los datos y gráficos obtenidos, y revisar que los resultados concuerden con los esperado: la ruta con debug por console.log es menos performante (bloquea más el servidor) que la que no lo contiene.

LOGGERS, GZIP y ANÁLISIS DE PERFORMANCE

Retomemos nuestro trabajo para implementar compresión por Gzip, registros por loggueo, y analizar la performance de nuestro servidor.

CODER HOUSE

LOGGERS Y GZIP

Formato: link a un repositorio en Github con el proyecto cargado.
Sugerencia: no incluir los node_modules

Desafío entregable

>> Consigna:

Incorporar al proyecto de servidor de trabajo la compresión gzip.

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

Luego implementar loggueo (con alguna librería vista en clase) que registre lo siguiente:

- Ruta y método de todas las peticiones recibidas por el servidor (info)
- Ruta y método de las peticiones a rutas inexistentes en el servidor (warning)
- Errores lanzados por las apis de mensajes y productos, únicamente (error)

Considerar el siguiente criterio:

- Loggear todos los niveles a consola (info, warning y error)
- Registrar sólo los logs de warning a un archivo llamada warn.log
- Enviar sólo los logs de error a un archivo llamada error.log

ANÁLISIS COMPLETO DE PERFORMANCE

Formato: link a un repositorio en Github con el proyecto cargado.
Sugerencia: no incluir los node_modules

Desafío entregable

>> Consigna: Luego, realizar el análisis completo de performance del servidor con el que venimos trabajando.

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:

1) 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.

ANÁLISIS COMPLETO DE PERFORMANCE

Formato: link a un repositorio en Github con el proyecto cargado.
Sugerencia: no incluir los node_modules

Desafío entregable

>> Consigna:

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)

2) 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.

3) El diagrama de flama con 0x, emulando la carga con Autocannon con los mismos parámetros anteriores.

Realizar un informe en formato pdf sobre las pruebas realizadas incluyendo los resultados de todos los test (texto e imágenes).

👉 Al final incluir la conclusión obtenida a partir del análisis de los datos.

¿PREGUNTAS?

¡MUCHAS GRACIAS!

Resumen de lo visto en clase hoy:

- Artillery y Autocannon como dos formas para realizar test de carga.
- Profiling con Node built-in Profiler.
- 0x para análisis de gráficos de flama interactivos basados en Autocannon.

OPINA Y VALORA ESTA CLASE

#DEMOCRATIZANDOLAEDUCACIÓN

CODER HOUSE