**Primer consigna**

1. **Compression:**

Implemento como middleware de aplicación el uso de la librería **compression** de la siguiente manera:

app.use(compression())

Al hacer esto, todos los llamados a las rutas serán comprimidas antes de ser devueltas.

Lo pruebo en la ruta api/info como pide la consigna pero no tiene sentido ya que es muy pequeño el paquete de información que se envía en esta petición. Tanto con como sin compresión el paquete pesa 1.1 kb.

Para que la prueba valga la pena hago lo mismo en la ruta api/randoms la cual es más pesada y obtengo los siguientes resultados:

Con compresión: **4.9 kb**

Sin compresión: **13.1 kb**

Demostrando, de esta manera, que la librería compression es útil y está siendo utilizada por todas mis rutas al pasarla como middleware de aplicación.

1. **Logger:**

Para la consigna de logger elegí utilizar la librería de **Winston**. Cree las reglas de logueos como piden las consignas y se pueden probar ingresando en la ruta **/api/productos/:id** con un id de producto no valido, por ejemplo, **62dc6faea37be7f856**. Esto genera un logueo tanto en consola como en el archivo **error.log** (adjunto en el repo).

Otra forma de probarlo es en la ruta **/api/productos** intentar de postear con el formulario un producto con datos incompletos, esto también devolverá un logueo de error tanto en consola como en el archivo error.log.

También están creadas las reglas de logueos de nivel **“warn”** y nivel **“info”**.

**Segunda Consigna:**

**1)**

La segunda consigna nos pide realizar un análisis de performance del servidor sobre la ruta **/api/info** agregando y quitando un console.log con la info que enviamos a la vista de esta ruta. Esto es para poder tener un proceso con tareas **bloqueantes** (la que tiene el console.log) y **no-bloqueantes.**

Todas las pruebas deben realizarse en el servidor en modo fork.

Primero inicializo el servidor agregando en console.log normalmente con el siguiente comando:

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

Y luego corro el primer test de artillery con: **artillery quick -c 20 -n 50 "http://localhost:8080/api/info" > result\_info\_Block.txt**

Luego elimino el console log del controller de info para ejecutar nuevamente el test de artillery: **artillery quick -c 20 -n 50 "http://localhost:8080/api/info" > result\_info\_NoBlock.txt**

Lo que puedo ver en este primer test es que los tiempos de promedios para atender cada request con el console.log (proceso bloqueante) son superiores que los tiempos promedios para atender los request sin el console.log (procesos no bloqueantes).

Luego levanto el servidor con el modo profilling de node de la siguiente forma ejecútandolo 2 veces. Una para el modo bloqueante y una para el no bloqueante:

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

Ejecuto nuevamente el commando artillery usando el siguiente comando:

Para modo bloqueante:

**artillery quick -c 20 -n 50 "http://localhost:8080/api/info" > result\_info\_Bloq.txt**

Para el modo no bloqueante:

**artillery quick -c 20 -n 50 "http://localhost:8080/api/info" > result\_info\_NoBloq.txt**

Luego mato el servidor de la consola y transformo el archivo de logs en un txt legible para el humano mediante los comandos:

**node.exe --prof-process noBloq-v8.log > result\_profNoBloq.txt**

y

**node.exe --prof-process bloq-v8.log > result\_profBloq.txt**

Donde puedo ver que el código es 100% JS, donde se ejecuta cada proceso (es decir los archivos donde están escritos), entre otras cosas.

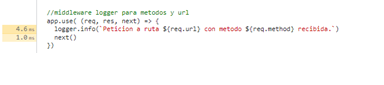
2) Ejecución de servidor en modo **--Inspect**:

Ejecuto el servidor con el siguiente comando para abrirlo en modo inspect:

**node –inspect src/server.js -p 8080 -m fork**

Luego voy a la herramienta de Chrome/inspect y starteo el análisis. Luego hago un nuevo test del servidor con artillery y analizo la herramienta de Chrome.

Llego a la siguiente conclusión:

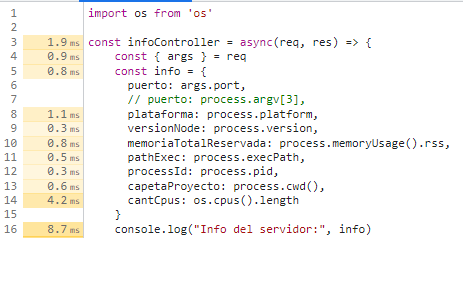
**El logger.info es una de las tareas que mas tiempo de ejecución consumen como puedo observar en esta captura:**  

**También, la necesidad de pasar los args dentro del req en todas las consultas de servidor tiene un gran consumo de recursos como se puede ver en la siguiente captura:**

Texto

Descripción generada automáticamente

**Por último, si vamos a ver en controller de infoController.js veremos que la mayor parte del tiempo del proceso se la lleva la cantCPus y luego también el console.log que hace que la tarea sea bloqueante:**



Esto tiene mucho sentido ya que ambos procesos son usados por el test de artillery sobre la ruta /api/info.

**3)** Por ultimo realizo un test con autocannon exigiéndole al servidor 100 conexiones durante 20 seg y estos son los resultados:

Para el test **bloqueante**, es decir, con console.log:

Interfaz de usuario gráfica

Descripción generada automáticamente

Para el test **no bloqueante**:

Interfaz de usuario gráfica

Descripción generada automáticamente

Como se puede observar, en el caso del test con console.log vemos que puede procesar **muchas menos request por segundo** (351 vs 624.9) ya que además tiene que enviar los console.log a la consolar del servidor. Esto hace también que tenga menos bytes descargados por segundo.

Como ultima conclusión de esto podemos ver que en el test bloqueante se procesaron 7k de request mientras que en el no bloqueante 13k de request, ambos en el mismo periodo de tiempo estipulado por autocannon: 20 seg.

Adjunto los gráficos de flama en el repositorio con sus html para ser consultados. Está hecho el gráfico tanto para el caso con el console.log como el caso sin este.

**Mi conclusión final es que en el gráfico no puede apreciarse demasiado la diferencia entre ambos procesos ya que son muy similares en cuanto a la topografía de los mismos, pero creo que tanto el reporte en .txt de artillery como el reporte por consola de autocannon son las mejores herramientas para poder estudiar el comportamiento del servidor y ver cuando un proceso es más performante que otro.**