



Clase 31. Programación Backend

Logs, profiling & debug ***Parte I***



OBJETIVOS DE LA CLASE

- Definir los loggers y sus características.
- Presentar las librerías más utilizadas para realizar loggers.
- Conocer las particularidades del rendimiento de las aplicaciones Node en producción y qué hacer para optimizarlo.

CRONOGRAMA DEL CURSO

Clase 30



PROXY & NGINX

Clase 31



**Logs, profiling & debug
Parte I**

Clase 32



**Logs, profiling & debug
Parte II**

RENDIMIENTO EN PRODUCCIÓN



NO SIRVE DE MUCHO REALIZAR APLICACIONES SUMAMENTE COMPLEJAS SI NO LOS USUARIOS LUEGO NO PODRÁN USARLAS.

El rendimiento y el consumo de recursos resulta una preocupación clave. Buscamos siempre generar un rendimiento óptimo con un tiempo de inactividad mínimo.

¿Qué podemos hacer en el código y qué en el entorno/configuración para optimizar nuestra aplicación?

COSAS QUE HACER EN EL CÓDIGO

Utilizar la compresión de gzip



- Una de las cosas que podemos hacer en el código para mejorar el rendimiento de una aplicación Express al desplegarla en producción es utilizar la **compresión de gzip**.
- Su uso puede disminuir significativamente el tamaño del cuerpo de respuesta y, por lo tanto, aumentar la velocidad de una aplicación web. Utilizamos **gzip**, un **middleware de compresión** de Node para la compresión en aplicaciones Express.
- Atención: **No** resulta la mejor opción para una aplicación con tráfico elevado en producción.

```
const express = require('express')
const compression = require('compression')
const app = express()
app.use(compression())
```


No utilizar funciones síncronas



- Las funciones síncronas y los métodos impiden el avance del proceso de ejecución hasta que vuelven.
- Una llamada individual a una función síncrona puede volver en pocos microsegundos, aunque en sitios web de tráfico elevado, estas llamadas se suman y reducen el rendimiento de la aplicación.
- **Evite su uso en producción.** Para esto, utilizar funciones asíncronas.
- La única vez que está justificado utilizar una función síncrona es en el arranque inicial.

Realizar un registro correcto



- El uso de **`console.log()`** o **`console.err()`** para imprimir mensajes de registro en el terminal es una práctica común en el desarrollo. No obstante, estas funciones son síncronas cuando el destino es un terminal o un archivo. De este modo, no resultan adecuadas para producción, a menos que canalice la salida a otro programa.
- En general, hay dos motivos para realizar un registro desde la aplicación:
 - **A efectos de depuración:** en lugar de utilizar `console.log()`, utilice un módulo de depuración especial como debug.
 - **Para registrar la actividad de la aplicación** (básicamente, todo lo demás): en lugar de utilizar `console.log()`, utilice una biblioteca de registro como Winston o Bunyan.

Manejar las excepciones correctamente



- Las aplicaciones Node se bloquean cuando encuentran una excepción no capturada. Si no manejamos las excepciones ni realizamos las acciones necesarias, la aplicación Express se bloqueará y quedará fuera de línea.
- Si seguimos el consejo de asegurarnos de que la aplicación se reinicia automáticamente más abajo, esta se recuperará de un bloqueo.
- Las aplicaciones Express normalmente necesitan un breve tiempo de arranque. Igualmente, deseamos evitar el bloqueo en primer lugar y, para ello, deberemos manejar correctamente las excepciones.
- Para asegurarnos de manejarlas todas y de forma correcta podemos usar **Try/Catch** y **Promises**.

COSAS QUE HACER EN EL ENTORNO/CONFIGURACIÓN

Establecer `NODE_ENV` en producción



- La variable de entorno `NODE_ENV` especifica el entorno en el que se ejecuta una aplicación (normalmente, desarrollo o producción). Una de las cosas más sencillas que puede hacer para mejorar el rendimiento es establecer ***`NODE_ENV` en producción***. Puede mejorarlos hasta 3 veces.
- Al establecerlo, Express almacenar en caché las plantillas de vistas y los archivos CSS generados y genera menos mensajes de error detallados.
- Si necesitamos escribir código específico del entorno, podemos comprobar el valor de `NODE_ENV` con `process.env.NODE_ENV`.
- Tener en cuenta que comprobar el valor de una variable de entorno supone una reducción de rendimiento, por lo que debe hacerse de forma moderada.

Que la App se reinicia automáticamente



- En producción, no deseamos que la aplicación esté fuera de línea en ningún momento. Esto significa que debe asegurarse de que se reinicia si la aplicación o el servidor se bloquean.
- Aunque esperamos que no se produzca ninguno de estos sucesos, si somos realistas, debemos tener en cuenta ambas eventualidades de la siguiente manera:
 - Utilizando un gestor de procesos para reiniciar la aplicación (y Node) cuando se bloquea.
 - Utilizando el sistema init que proporciona su OS para reiniciar el gestor de procesos cuando se bloquea el OS.

Ejecutar la App en un Cluster



- En un sistema multinúcleo, podemos multiplicar el rendimiento de una aplicación Node iniciando un clúster de procesos.
- Como ya vimos, un clúster ejecuta varias instancias de la aplicación, idealmente una instancia en cada núcleo de CPU, lo que permite distribuir la carga y las tareas entre las instancias.
- En las aplicaciones en clúster, los procesos worker pueden bloquearse individualmente sin afectar al resto de los procesos.
- Aparte de las ventajas de rendimiento, el aislamiento de errores es otro motivo para ejecutar un clúster de procesos de aplicación. Siempre que se bloquee un proceso worker, hay que asegurarse de registrar el suceso y generar un nuevo proceso utilizando *cluster.fork()*.

Almacenar en caché los resultados de la solicitud



- Otra estrategia para mejorar el rendimiento en producción es almacenar en caché el resultado de las solicitudes, para que la aplicación no repita la operación de dar servicio a la misma solicitud repetidamente.
- Utilizar un servidor de almacenamiento en memoria caché como Nginx, mejora significativamente la velocidad y el rendimiento de la aplicación.

Utilizar el balanceador de carga

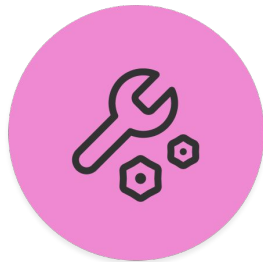


- Una única instancia sólo puede manejar una cantidad limitada de carga y tráfico. Una forma de escalar una aplicación es ejecutar varias instancias de la misma y distribuir el tráfico utilizando un balanceador de carga.
- Un balanceador de carga, como ya vimos, es un proxy inverso que orquesta el tráfico hacia y desde los servidores y las instancias de aplicación.
- Entonces, con el balanceador de carga, configurado por ejemplo con Nginx, podemos mejorar el rendimiento y velocidad de la aplicación permitiendo escalarla más que con una sola instancia

Utilizar un proxy inverso



- Como ya vimos, un proxy inverso se coloca delante de una aplicación web y realiza operaciones de soporte en las solicitudes, aparte de dirigir las solicitudes a la aplicación. Puede manejar las páginas de errores, la compresión, el almacenamiento en memoria caché, el servicio de archivos y el equilibrio de carga, entre otros.
- La entrega de tareas que no necesitan saber el estado de la aplicación a un proxy inverso permite a Express realizar tareas de aplicación especializadas. Por este motivo, se recomienda ejecutar Express detrás de un proxy inverso como Nginx en producción.



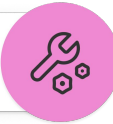
PROBANDO GZIP

Tiempo: 8 minutos

Probando Gzip

Tiempo: 8 minutos

Desafío
generico



Realizar un servidor con dos endpoints GET, cada uno que devuelva la frase 'Hola que tal' concatenada 1000 veces, en las rutas '/saludo' y '/saludozip'.

Al manejador de '/saludozip' agregar gzip como middleware.

Probar ambos endpoints y verificar en el navegador cuántos bytes llegan como respuesta desde el servidor y qué headers trae la respuesta.

Sabiendo que 1000 veces 12 caracteres de 1 byte c/u equivale a 12000 bytes (~12kb) ese es tamaño de paquete que esperamos recibir. Chequear si es así en cada caso.



BREAK

¡5/10 MINUTOS Y VOLVEMOS!

LOGGERS



¿Qué son?



- Cuando llevamos un sistema a producción, uno de los elementos más importantes a la hora de detectar cualquier problema o anomalía son los logs.
- Cuando hay muchas peticiones concurrentes, los **logs** de todas ellas se mezclan haciendo imposible su seguimiento salvo que tengan un **identificador único**.



Un log o historial de log refiere al registro secuencial de cada uno de los eventos que afectan un proceso particular constituyendo una evidencia del comportamiento del sistema.



¿Qué son?



- Los loggers son librerías para facilitar la impresión de un identificador único.
- Tienen la ventaja de que no necesitamos usar *console.log* para el registro de sucesos en el servidor, lo cual es más eficiente y permiten clasificar los mensajes por niveles de *debug* y enviarlos a distintos medios: file, database, email, consola, etc.

LIBRERÍA LOG4JS

Usando Log4js

Es una de las librerías de loggers más utilizada. Aunque actualmente está siendo reemplazada por Winston y luego por Pino, que es hoy el más moderno.

1. Para empezar a utilizarlo, primero debemos instalarlo:

```
$ npm install log4js
```
2. Luego, lo requerimos en el *app.js* o archivo principal de la aplicación.

```
const log4js = require("log4js");
```



Usando Log4js

3. Luego, debemos configurarlo, mediante el siguiente código.

```
log4js.configure({
  appenders: {
    miLoggerConsole: { type: "console" },
    miLoggerFile: { type: 'file', filename: 'info.log' },
    miLoggerFile2: { type: 'file', filename: 'info2.log' }
  },
  categories: {
    default: { appenders: ["miLoggerConsole"], level: "trace" },
    consola: { appenders: ["miLoggerConsole"], level: "debug" },
    archivo: { appenders: ["miLoggerFile"], level: "warn" },
    archivo2: { appenders: ["miLoggerFile2"], level: "info" },
    todos: { appenders: ["miLoggerConsole", "miLoggerFile"], level:
"error" }
  }
})
```

- Definimos 3 apéndices:
miLoggerConsole usa el apéndice stdout escribe en la salida estándar (consola). Los otros 2, usan el archivo adjunto. **miLoggerFile** escribe en el archivo *info.log* y **miLoggerFile2** en el archivo *info2.log*.

Usando Log4js



```
log4js.configure ({
  appenders: {
    miLoggerConsole: { type: "console" },
    miLoggerFile: { type: 'file', filename: 'info.log' },
    miLoggerFile2: { type: 'file', filename: 'info2.log' }
  },
  categories: {
    default: { appenders: ["miLoggerConsole"], level: "trace" },
    consola: { appenders: ["miLoggerConsole"], level: "debug" },
    archivo: { appenders: ["miLoggerFile"], level: "warn" },
    archivo2: { appenders: ["miLoggerFile2"], level: "info" },
    todos: { appenders: ["miLoggerConsole", "miLoggerFile"], level:
"error" }
  }
})
```

- Definimos 5 categorías con distintos niveles:
 - Las categorías *default* y *consola* utilizan el apéndice del tipo *console*.
 - Las categorías *archivo* y *archivo2* utilizan los apéndices de tipo *file*.
 - La categoría *todos* utiliza apéndice de tipo *console* y tipo *file*.

Niveles de salida y ventajas



- Definimos 6 niveles de salida: Trace, Debug, Info, Warn, Error, Fatal.
- Los niveles que se imprimen, son desde el especificado en la configuración de categorías para abajo. Por ejemplo, si el nivel configurado es Warn, se imprimirá solo Warn, Error y Fatal.
- La ventaja de esto es que en un entorno de producción podemos solo preocuparnos por las excepciones y errores y no por la información de depuración.
- Otra ventaja es que el código se puede mezclar con varios códigos de impresión de registros. Siempre que modifiquemos el nivel de salida en un archivo de configuración, la salida del registro cambiará sin modificar todo el código.

Niveles de salida y categorías



Ejecutando la categoría **default**

```
const logger = log4js.getLogger();  
logger.trace("Entering cheese testing");  
logger.debug("Got cheese.");  
logger.info("Cheese is Comté.");  
logger.warn("Cheese is quite smelly.");  
logger.error("Cheese is too ripe!");  
logger.fatal("Cheese was breeding ground for listeria.");
```

En consola se imprime

```
[2021-05-02T11:32:16.906] [TRACE] default - Entering cheese testing  
[2021-05-02T11:32:16.909] [DEBUG] default - Got cheese.  
[2021-05-02T11:32:16.909] [INFO] default - Cheese is Comté.  
[2021-05-02T11:32:16.909] [WARN] default - Cheese is quite smelly.  
[2021-05-02T11:32:16.910] [ERROR] default - Cheese is too ripe!  
[2021-05-02T11:32:16.910] [FATAL] default - Cheese was breeding ground for listeria.
```

Niveles de salida y categorías



Ejecutando la categoría **consola**

```
const loggerConsola = log4js.getLogger('consola');  
loggerConsola.trace("Entering cheese testing");  
loggerConsola.debug("Got cheese.");  
loggerConsola.info("Cheese is Comté.");  
loggerConsola.warn("Cheese is quite smelly.");  
loggerConsola.error("Cheese is too ripe!");  
loggerConsola.fatal("Cheese was breeding ground for listeria.");
```

En consola se imprime

```
[2021-05-02T11:35:15.464] [DEBUG] consola - Got cheese.  
[2021-05-02T11:35:15.467] [INFO] consola - Cheese is Comté.  
[2021-05-02T11:35:15.468] [WARN] consola - Cheese is quite smelly.  
[2021-05-02T11:35:15.468] [ERROR] consola - Cheese is too ripe!  
[2021-05-02T11:35:15.469] [FATAL] consola - Cheese was breeding ground for listeria.
```

Niveles de salida y categorías



Ejecutando la categoría ***archivo***

```
const loggerArchivo = log4js.getLogger('archivo');  
loggerArchivo.trace("Entering cheese testing");  
loggerArchivo.debug("Got cheese.");  
loggerArchivo.info("Cheese is Comté.");  
loggerArchivo.warn("Cheese is quite smelly.");  
loggerArchivo.error("Cheese is too ripe!");  
loggerArchivo.fatal("Cheese was breeding ground for listeria.");
```

En el archivo *info.log*
se imprime

```
[2021-05-02T11:36:44.281] [WARN] archivo - Cheese is quite smelly.  
[2021-05-02T11:36:44.283] [ERROR] archivo - Cheese is too ripe!  
[2021-05-02T11:36:44.283] [FATAL] archivo - Cheese was breeding ground for listeria.
```


Niveles de salida y categorías



Ejecutando la categoría ***archivo2***

```
const loggerArchivo2 = log4js.getLogger('archivo2');  
loggerArchivo2.trace("Entering cheese testing");  
loggerArchivo2.debug("Got cheese.");  
loggerArchivo2.info("Cheese is Comté.");  
loggerArchivo2.warn("Cheese is quite smelly.");  
loggerArchivo2.error("Cheese is too ripe!");  
loggerArchivo2.fatal("Cheese was breeding ground for listeria.")
```

En el archivo
info2.log se imprime

```
[2021-05-02T11:38:39.013] [INFO] archivo2 - Cheese is Comté.  
[2021-05-02T11:38:39.014] [WARN] archivo2 - Cheese is quite smelly.  
[2021-05-02T11:38:39.015] [ERROR] archivo2 - Cheese is too ripe!  
[2021-05-02T11:38:39.015] [FATAL] archivo2 - Cheese was breeding ground for listeria.
```

Niveles de salida y categorías



Ejecutando la categoría ***todos***

```
const loggerTodos = log4js.getLogger('todos');  
loggerTodos.trace("Entering cheese testing");  
loggerTodos.debug("Got cheese.");  
loggerTodos.info("Cheese is Comté.");  
loggerTodos.warn("Cheese is quite smelly.");  
loggerTodos.error("Cheese is too ripe!");  
loggerTodos.fatal("Cheese was breeding ground for listeria.");
```

En consola se imprime

```
[2021-05-02T11:40:49.333] [ERROR] todos - Cheese is too ripe!  
[2021-05-02T11:40:49.337] [FATAL] todos - Cheese was breeding ground for listeria.
```

En el archivo *info.log*
se imprime

```
[2021-05-02T11:40:49.333] [ERROR] todos - Cheese is too ripe!  
[2021-05-02T11:40:49.337] [FATAL] todos - Cheese was breeding ground for listeria.
```

Niveles de salida y categorías



Es posible definirle a un appender directamente un nivel para que loguee usando ese criterio siempre, independientemente de la categoría que lo use. Esto nos permite, por ejemplo, definir una categoría que loguee en un archivo todo lo que sea nivel info, y loguee por consola todos los errores. Para esto, debemos crear dentro de los appenders un ítem especial que defina dichos criterios:

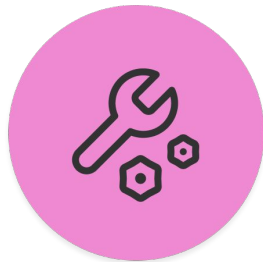
```
appenders: {  
  // defino dos soportes de salida de datos  
  consola: { type: 'console' },  
  archivo: { type: 'file', filename: 'errores.log' },  
  // defino sus niveles de logueo  
  loggerConsola: { type: 'logLevelFilter', appender: 'consola', level: 'info' },  
  loggerArchivo: { type: 'logLevelFilter', appender: 'archivo', level: 'error' }  
},
```

Niveles de salida y categorías



Luego al definir las categorías, defino una que utilice más de un appender. Es importante que al momento de utilizar appenders con niveles personalizados, definamos el nivel de la categoría como 'all', para que permita los distintos valores que definimos anteriormente.

```
categories: {  
  default: {  
    appenders: ['loggerConsola'], level: 'all'  
  },  
  custom: {  
    appenders: ['loggerConsola', 'loggerArchivo'], level: 'all'  
  }  
}
```



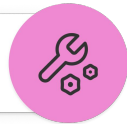
LOGUEAR CON LOG4JS

Tiempo: 10 minutos

Loguear con Log4js

Tiempo: 10 minutos

Desafío
generico



Crear un servidor que tenga una ruta '/sumar' que reciba por query params dos números y devuelva un mensajes con la suma entre ambos.

Utilizar log4js para crear un módulo capaz de exportar uno de los siguientes dos loggers: uno para el entorno de desarrollo, que logueará de info en adelante por consola, y otro para el entorno de producción, que logueará de debug en adelante a un archivo 'debug.log' y solo errores a otro archivo 'errores.log'.

Loguear con Log4js

Tiempo: 10 minutos

Desafío
generico



El logueo se realizará siguiendo el siguiente criterio:

- En caso de operaciones exitosas, loguear una línea de info
- En caso de ingresar un número no válido, loguear un error
- En caso de fallar el inicio del servidor, loguear un error
- En caso de recibir una petición a un recurso inválido, loguear una warning.

La decisión de qué logger exportar se tomará en base al valor de una variable de entorno `NODE_ENV`, cuyo valor puede ser: 'PROD' para producción, o cualquier otra cosa (incluyendo nada) para desarrollo.

LIBRERÍA WINSTON

¿Qué es Winston?



- Winston es una librería con soporte para múltiples transportes diseñada para el registro simple y universal.
- Un *transporte* es esencialmente un dispositivo que nos permiten almacenar mensajes personalizados de seguimiento (al igual que *console.log*) en un archivo plano o desplegado por consola.
- Cada logger de Winston puede tener múltiples transportes, configurados en diferentes niveles.

Usando Winston



1. Para empezar a utilizarlo, primero debemos instalarlo:
2. Requerimos el paquete Winston y lo configuramos:

```
$ npm install winston
```

```
const winston = require('winston')

const logger = winston.createLogger({
  level: 'warn',
  transports : [
    new winston.transports.Console({ level:'verbose' }),
    new winston.transports.File({ filename: 'info.log', level:'error' }),
  ]
})
```

3. Dentro del método **winston.createLogger** definimos primero el nivel de registro que vamos a desplegar.
4. Luego, en este caso definimos 2 transportes. Uno en el nivel *verbose* que escribe en consola y otro en nivel *error* que escribe en el archivo *info.log*.

Niveles de salida



- Los niveles de salida definidos en Winston son: Silly, Debug, Verbose, Info, Warn, Error.
- Al igual que en Log4js, se imprime desde el nivel especificado hacia los niveles con mayor prioridad (los anteriores no se imprimen).
- Se puede imprimir con el siguiente código, especificando el nivel de salida y el mensaje que se desea imprimir.
- Con este código, se va a imprimir en todos los transportes (en el caso que configuramos antes, sería en consola y en el archivo:

```
logger.log('level', 'message')
```

Niveles de salida y transportes



- Ejecutando entonces los dos transportes con cada uno de los niveles de salida:

```
logger.log('silly', "127.0.0.1 - log silly")
logger.log('debug', "127.0.0.1 - log debug")
logger.log('verbose', "127.0.0.1 - log verbose")
logger.log('info', "127.0.0.1 - log info")
logger.log('warn', "127.0.0.1 - log warn")
logger.log('error', "127.0.0.1 - log error")
```

- Por consola se imprime:

```
{"level":"verbose","message":"127.0.0.1 - log verbose"}
{"level":"info","message":"127.0.0.1 - log info"}
{"level":"warn","message":"127.0.0.1 - log warn"}
{"level":"error","message":"127.0.0.1 - log error"}
```

- En el archivo info.log se imprime:

```
{"level":"error","message":"127.0.0.1 - log error"}
```

👉 Como podemos observar, solo se imprimen los de niveles de salida que sean el configurado en el transporte y los que siguen en prioridad.

Niveles de salida y transportes



También podemos ejecutarlo de la siguiente manera:

- Se ejecutan también todos los transportes configurados:
- Por consola se imprime:
- En el archivo info.log se imprime:

```
logger.info("127.0.0.1 - log info 2")  
logger.warn("127.0.0.1 - log warn 2")  
logger.error("127.0.0.1 - log error 2")
```

```
{"message":"127.0.0.1 - log info 2","level":"info"}  
{"message":"127.0.0.1 - log warn 2","level":"warn"}  
{"message":"127.0.0.1 - log error 2","level":"error"}
```

```
{"message":"127.0.0.1 - log error 2","level":"error"}
```

👉 Al igual que antes, solo se imprimen los del nivel de salida configurado y los que siguen en prioridad (por eso en el archivo solo se imprime el error, ya que es el nivel configurado en su transporte).



LOGUEAR CON WINSTON

Tiempo: 5 minutos

Loguear con Winston

Tiempo: 5 minutos

Desafío
generico



Realizar el ejercicio anterior pero esta vez utilizando winston logger.

Crear los loggers respetando los niveles de log y las capas de transporte necesarias para cumplir con el enunciado.

LIBRERÍA PINO

Usando Pino



Pino es la librería más moderna de las utilizadas actualmente. Es veloz y cuenta con un buen ecosistema de trabajo.

Para empezar a utilizarlo:

1. Debemos instalarlo: `$ npm install pino`
2. Lo requerimos tal como muestra el siguiente código:
3. Seteamos su nivel:

```
const logger = require('pino')()
```

```
logger.level = 'info'
```



Instancias de Logger

- La instancia del Logger es el objeto devuelto por la función principal de Pino.
- Su propósito principal es proveer los **métodos de logging**.
- Los métodos por default son: Trace, Debug, Info, Warn, Error y Fatal.
- Todos los métodos tienen la siguiente forma genérica:

```
logger.method([mergingObject], [message], [...interpolationValues])
```

Instancias de Logger

MergingObject



Opcionalmente, se puede proporcionar un objeto como primer parámetro. Cada par clave valor enumerable del *mergingObject* se copia en la línea de log JSON.

Instancias de Logger

Message



- Se puede proporcionar opcionalmente, un string como parámetro. Por default, se fusiona en el *log JSON*, en la clave *msg*.
- Este parámetro tiene prioridad respecto al de *mergingObject*. Es decir, si *mergingObject* tiene un mensaje y además se especifica el parámetro *message*, el que se va a imprimir es el de *message*.
- Los string de message, pueden contener algún marcador de posición (placeholder). Estos son “%s” para string, “%d” para dígitos, “%0”, “%o” y “%j” para objetos. Los valores para estos marcadores de posición se proporcionan como un parámetro extra.

Instancias de Logger

InterpolationValues



- Todos los argumentos suministrados después del mensaje se serializan e interpolan de acuerdo con los marcadores de posición de estilo printf, suministrados para formar el valor de mensaje de salida final para la línea de *log JSON*.
- En el siguiente código, vemos que está solo el parámetro de `message`.

```
logger.info('pino info')  
logger.error('pino error')
```

- En este caso, en consola se imprime:

```
{"level":30,"time":1619999222171,"pid":256,"hostname":"LAPTOP-V5331C85","msg":"pino info"}  
{"level":50,"time":1619999222172,"pid":256,"hostname":"LAPTOP-V5331C85","msg":"pino error"}
```

👉 Vemos que los niveles de salida se imprimen con su código (30 para *info* y 50 para *error*).

Instancias de Logger

InterpolationValues



- En el siguiente código, en la primera línea usamos el marcador de posición “%d” para el número 42. En las siguientes líneas, utilizamos mergingObjects con distintos objetos.

```
logger.info('La respuesta es %d',42)
logger.info({a:42},'Hola mundo')
logger.info({a:42,b:2},'Hola mundo')
logger.info({c: {a:42,b:2}},'Hola mundo')
```

- En este caso, en consola se imprime:

```
{"level":30,"time":1619999533848,"pid":19704,"hostname":"LAPTOP-V5331C85","msg":"La respuesta es 42"}
{"level":30,"time":1619999533849,"pid":19704,"hostname":"LAPTOP-V5331C85","a":42,"msg":"Hola mundo"}
{"level":30,"time":1619999533849,"pid":19704,"hostname":"LAPTOP-V5331C85","a":42,"b":2,"msg":"Hola mundo"}
{"level":30,"time":1619999533850,"pid":19704,"hostname":"LAPTOP-V5331C85","c":{"a":42,"b":2},"msg":"Hola mundo"}
```



Método `logger.child`

- El método ***`logger.child`*** permite la creación de registradores con estado (*stateful loggers*), donde los pares clave-valor se pueden anclar a un *logger*, lo que hace que se generen en cada línea de *log*.
- Los *logger.child* usan el mismo flujo de salida que el padre y heredan el nivel de log actual del padre en el momento en que se generan.
- El nivel de registro de un *logger.child* es mutable. Se puede configurar independientemente del padre, ya sea configurando el acceso de nivel después de crear el *log* secundario o usando la clave reservada ***`bindings.level`***.
- Los *logger.child* heredan los serializadores del *log* principal.



Método *logger.child*

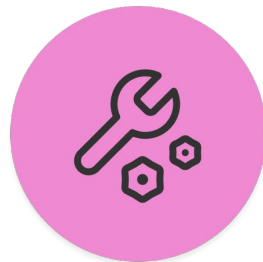
- En el siguiente código, vemos cómo podemos usar este método.

```
const child = logger.child({a: 'property'})  
child.info('Hola child info')  
child.info('Hola child info 2')  
child.error('Hola child error')
```

- En consola se imprime lo siguiente:

```
{ "level": 30, "time": 1620000279082, "pid": 5580, "hostname": "LAPTOP-V5331C85", "a": "property", "msg": "Hola child info" }  
{ "level": 30, "time": 1620000279083, "pid": 5580, "hostname": "LAPTOP-V5331C85", "a": "property", "msg": "Hola child info 2" }  
{ "level": 50, "time": 1620000279083, "pid": 5580, "hostname": "LAPTOP-V5331C85", "a": "property", "msg": "Hola child error" }
```

👉 Podemos observar que es muy similar a lo mostrado anteriormente. Sin embargo, en los 3 casos, se imprimió además, antes del parámetro de *message*, el objeto especificado en la primera línea, al definir el método *logger.child*.



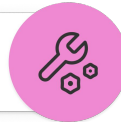
LOGUEAR CON PINO

Tiempo: 8 minutos

Loguear con Pino

Tiempo: 8 minutos

Desafío
generico



Realizar el ejercicio anterior pero esta vez utilizando pino logger. En el caso del logger para el entorno de producción, loguear a un único archivo 'debug.log', con nivel 'debug'.

(Pino no soporta actualmente la salida en simultáneo por más de un transporte)

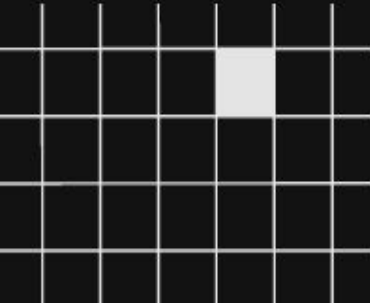
¿PREGUNTAS?





¡MUCHAS GRACIAS!

Resumen de lo visto en clase hoy:

- Mejoras para el rendimiento de la aplicación Express.
 - Loggers. Por qué son importantes y las librerías de npm más utilizadas para esto.
- 



OPINA Y VALORA ESTA CLASE

#DEMOCRATIZANDO LA EDUCACIÓN

CODER HOUSE