**Почему не меняется дефолтное значение { level: 'info' }**

**Настройки регистратора Pino в Fastify – часть 1.**

**Описание**

Какое-то время назад начал знакомиться с фреймворком Fastify. Достаточно интересный фреймворк с прекрасной документацией (но местами могло быть и лучше).

Особенность Fastify в том, что у него своя экосистема, и много о чём разработчики подумали за нас. При выборе инструментов для своей экосистемы, разработчики Fastify зачастую делали выбор в пользу более производительных (*быстрых*) решений. Так же получилось и с логированием. Логирование в Fastify работает из коробки с помощью регистратора (логгера) [**Pino**](https://github.com/pinojs/pino). О нём и поговорим.

Здесь не будет исчерпывающего гайда по настройке логирования, для этих целей существует документация. Прочтение данной статьи даст базу для самостоятельной настройки Pino как для Fastify, так и отдельно. Рассмотрим, как работает Pino под капотом и попробуем сконфигурировать Pino для разных сред (разработка, production).

**Краткий обзор Fastify и Pino**

[**Pino**](https://getpino.io/) позиционирует себя как Node.js регистратор с «очень низкими накладными расходами» (*very low overhead*). Здесь имеется в виду, что на логирование затрачивается меньше ресурсов процессора и памяти, а значит всё работает быстрее и как следствие – дешевле.

Если верить документации, в ряде тестов Pino показывает себя в 5 раз быстрее чем некоторые конкуренты (звучит как маркетинг), но в любом случае - вот [официальные замеры (benchmarks)](https://github.com/pinojs/pino/blob/master/docs/benchmarks.md). Как вариант, можете ознакомиться со статьёй [«The Cost of Logging in Modern Applications»](https://www.nearform.com/blog/the-cost-of-logging-in-2022/).

[**Fastify**](https://www.fastify.io/) позиционирует себя как веб-фреймворк, ориентированный на обеспечение наилучшего пользовательского опыта в разработчике с наименьшими накладными расходами и мощной архитектурой плагинов.

Интересный факт, что Node.js, Fastify и Pino разрабатывают одни и те же люди, особенно популярен [Matteo Collina](https://github.com/mcollina). На лицо ~~кумовство~~ очевидная ориентированность на лучшие практики Node.js под капотом.

Возможности логирования **Fastify** не ограничиваются одним лишь **Pino**. При желании, можно подключить к Fastify иные регистраторы, например [Winston](https://www.npmjs.com/package/winston). Но учитывайте, что в зависимости от выбора регистратора, возможно придётся помучаться с ~~костылями~~ конфигурацией, чтобы настройки вашего логгера прошли внутреннюю валидацию Fastify. Fastify совместима с логгерами, поддерживающими интерфейс [стандарта Log4j](https://en.wikipedia.org/wiki/Log4j).

По сути, достаточно чтобы регистратор поддерживал [определенные уровни (levels)](https://www.tutorialspoint.com/log4j/log4j_logging_levels.htm) логирования и кое-какие методы. В любом случае, в интернете можно найти пару готовых решений этой задачи разной степени свежести (я проверял!) или прокачать свои навыки написав решение с нуля.

Также, согласно документации Fastify, вы можете подключить внешний регистратор с помощью библиотеки [abstract-logging](https://www.npmjs.com/package/abstract-logging), которая также работает с регистраторами, поддерживающими интерфейс стандарта Log4j.

Возможности Pino очень обширны и покрывают практически все потребности. Pino позволяет выводить логи в консоль, сохранять их в локальные файлы, менять формат сообщений, предавать логи в самые разные базы данных, во внешние инструменты, такие как [Kibana (ELK)](https://www.elastic.co/kibana/), [Grafana Loki](https://grafana.com/oss/loki/), [Seq](https://datalust.co/seq), [Sentry](https://sentry.io/welcome/), [Airbrake](https://www.airbrake.io/), [Datadog](https://getpino.io/#/docs/transports?id=pino-datadog-transport), [Apache Kafka](https://kafka.apache.org/) и другие. Подробнее, про возможности интеграции (транспортировки логов) можете почитать в [соответствующем разделе документации Pino](https://getpino.io/#/docs/transports?id=known-transports).

На момент написания статьи, у Pino чуть более 3,9 млн. загрузок на GitHub, проект жив, у него много контрибьютеров и в англоязычных статьях очень тепло отзываются о Pino.

**Подключение Pino**

Pino работает в Fastify из коробки, а значит не нужно его отдельно устанавливать. Убедиться в этом можно заглянув в *package.json* самого фреймворка Fastify в папке */node\_modules/fastify/package.json.*

Включить логирование в Fastify очень легко. Достаточно при создании экземпляра Fastify-сервера передать в параметры функции { logger: true }).

```javascript

**// app.js**

import Fastify from 'fastify'

const app = Fastify({ logger: true })

```

Если ранее вы уже работали с Pino и у вас на руках остался код готового сконфигурированного экземпляра Pino, то вместо { logger: true }) вы можете передать в Fastify свой экземпляр Pino, как это показано ниже.

```javascript

// создаём экземпляр Pino с какими-то настройками

const log = require('pino')({ level: 'info' })

// передаём наш готовый экземпляр в Fastify

const fastify = require('fastify')({ logger: log })

```

По этой же логике, мы можем использовать иные регистраторы, например, мы [можем передать в Fastify экземпляр Winston](https://gist.github.com/gclem/40ac30dfaa002c4c250ed91f65f560ee).

**Как это работает на уровне кода**

Давайте ненадолго заглянем в папку node\_modules и посмотрим, что происходит, когда мы передаем в функцию Fastify объект `{ logger: true }` или сконфигурированный экземпляр логгера.

Для создания нового экземпляра Fastify-сервера используется «фабричная функция». Эта фабричная функция принимает объект options `(typeof options == 'object')`, который используется для настройки результирующего экземпляра сервера. Рассмотрим, как это реализовано на уровне кода *(см. файл /node\_modules/fastify/fastify.js).*

```javascript

**// fastify.js**

// Импортируем функцию createLogger, которая создаёт экземпляр Pino

// и возвращает { logger, hasLogger }, где logger – экземпляр Pino.

const { createLogger } = require('./lib/logger')

function fastify (options) {

// валидация options (должен быть объект)

options = options || {}

if (typeof options !== 'object') {

throw new TypeError('Options must be an object')

}

// если валидация прошла успешно, то вызываем функцию createLogger

// которая создаёт и возвращает экземпляр логгера Pino.

const { logger, hasLogger } = createLogger(options)

// обновляем параметры с фиксированными значениями

options.logger = logger

// создаём объект fastify, который содержит API сервера

const fastify = {

// API для подключения любого логгера из вне

log: logger

}

// возвращаем объект сервера fastify

return fastify

}

```

Функция createLogger(options) создаёт экземпляр Pino и возвращает его. *(см. файл /node\_modules/fastify/lib/logger.js):*

```javascript

**// *logger.js***

const pino = require('pino')

function createPinoLogger (opts, stream) {

// немного валидации opts и stream и логики,

// затем создаём logger со значением null

let logger = null

// валидация, связанная с версиями библиотеки, и если всё нормально

// создам экземпляр Pino и записываем его в переменную logger

if (/\*…\*/) {

logger = pino(opts, stream)

}

// возвращаем logger, либо с значением null, либо как экземпляр Pino

return logger

}

// Функция createLogger внутри себя либо вызывает createPinoLogger и

// создаёт экземпляр Pino и экспортирует его для конфигурирования

// создаваемого экземпляра Fastify-сервера

function createLogger (options) {

// валидация options

if (isValidLogger(options.logger)) {

// создание экземпляра Pino

const logger = createPinoLogger({

logger: options.logger,

serializers: Object.assign({}, serializers, options.logger.serializers)

})

//возвращает экземпляр логгера

return { logger, hasLogger: true }

} else if (!options.logger) {

// иная логика

return { logger, hasLogger: false }

} else {

// иная логика

return { logger, hasLogger: true }

}

}

```

Теперь мы на базовом уровне понимаем, что когда при создании экземпляра Fastify-сервера мы передаем в него объект options { logger: true } , то Fastify под капотом создаёт экземпляр логгера Pino c настройками по умолчанию и возвращает экземпляр Fastify-сервера с дефолтным Pino.

В то же время, Fastify проверяет, не является ли объект options готовым сконфигурированным экземпляром какого-нибудь логгера. Это может быть тот же [Pino](https://github.com/pinojs/pino) или любой другой логгер, например [Winston](https://www.npmjs.com/package/winston), [Bunyan](https://www.npmjs.com/package/bunyan), [Morgan](https://www.npmjs.com/package/morgan) и др. Главное, чтобы логгер был настроен надлежащим образом и соответствовал стандартному интерфейсу Log4j.

**Использование Pino в разных средах (dev, prod, test)**

Пока мы не начали конфигурировать Pino, рассмотрим, как в принципе разделить конфигурацию для применения в разных средах. Логирование в продакшене (prod), разработке (dev) и тестировании (test) должно решать разные задачи. Например:

- **в разработке** я хочу, чтобы Pino перехватывал все возможные события. Чтобы логи красиво выводились в консоль с использованием разных цветов в зависимости от типа события. Так же хочу, чтобы сообщения в консоли выводились в удобном для чтения формате. Помимо консоли, я хочу, чтобы логи сохранялись в какой-нибудь файл в json-формате.

- **в production** я хочу, чтобы Pino перехватывал только некоторые действия пользователей и все ошибки. Мне не нужно чтобы в консоль что-то выводилось. Я хочу сохранять логи в разные файлы, в зависимости от типа события.

Также, я хочу отдельно отправлять логи стороннее приложение. Например, в Kibana (ELK).

- **в тестировании** мне не нужно логирование.

> **Внимание!**

> Фраза «в тестировании мне не нужно логирование» – это **просто для примера**! Не нужно воспринимать эту фразу как инструкцию к применению. Нужно вам логирование в тестировании или нет вы решаете сами исходя из ваших задач.

Для начала, вынесем конфигурацию Pino в отдельный файл. Создадим в корне приложения папку pino со следующей структурой:

```

pino/

config/

config.js

dev.js

prod.js

test.js

logs/

dev.log

```

Содержание файла ***config.js*:**

``` javascript

// импортируем конфигурации для каждой среды

const { default: dev } = await import('./dev.js')

const { default: prod } = await import('./prod.js')

const { default: test } = await import('./test.js')

// объединим конфигурацию для разных сред в единый объект

// и экспортируем его для использования в Fastify()

export default {

dev,

prod,

test,

}

```

Пока вместо реальной конфигурации поставим «заглушки» и пропишем в файлах ***dev.js*** и ***prod.js*** следующее**:**

``` javascript

export default {

transport: {

// наша конфигурация

},

}

```

Поскольку в данном случае логирование во время тестирования нам не нужно, то в файле ***test.js*** пропишем**:**

``` javascript

export default false // значение false отключит логирование

**```**

Затем, просто импортируем наш объект конфигурации из файла ***config.js*** вкорневой файл приложения ***app.js*** и передадим его в Fastify с нужным нам ключом:

```javascript

**// app.js**

import Fastify from 'fastify'

const { default: pino } = await import(‘./pino/config/config.js’)

//для разработки

const fastify = Fastify({ logger: pino.dev })

// для production среды

// const fastify = Fastify({ logger: pino.prod })

```

Я думаю, что логика разделения конфигурации понятна. При желании, можно добавить немного валидации, которая будет возвращать ту или иную конфигурацию в зависимости от переменных окружения., т.е. в зависимости от того, в какой среде будет выполняться код.

**Немного про уровни (levels) логирования**

Возможные значения параметра level – `'fatal'`, `'error'`, `'warn'`, `'info'`, `'debug'`, `'trace'` или `'silent'`. **По умолчанию**: `{ level: 'info' }`.

Уровни в Pino по сути являются реализацией стандартного интерфейса Log4j. Подробнее про Log4j и про уровни, можно прочитать, например [на википедии](https://en.wikipedia.org/wiki/Log4j#Log4j_log_levels), а если конкретно про уровни – то например [вот здесь](https://www.tutorialspoint.com/log4j/log4j_logging_levels.htm).

Level – это минимальный уровень для записи в журнал, т.е. Pino не будет регистрировать сообщения с более низким уровнем. Установка этой опции **снижает нагрузку**, так как обычно журналы отладки (debug) и трассировки (trace) используются только для разработки и не нужны в производстве.

При этом мы можем создать и использовать свои пользовательские (custom) уровни. Дополнительные уровни могут быть добавлены к экземпляру с помощью опции [customLevels](https://getpino.io/#/docs/api?id=level-string).

Установка минимального уровня журнала обычно выполняется при создании экземпляра регистратора и управляется через переменную окружения, чтобы можно было изменять его в различных средах без внесения изменений в код.

```javascript

prod: {

transport: {

target: ‘pino/file’

options: {

level: process.env.PINO\_LOG\_DEV\_LEVEL || 'debug',

},

},

},

```

**Настройка логирования для разработки**

В нашем случае, при настройке логирования мы должны решить 2 основные задачи:

- одновременно выводить логи в консоль и сохранять логи в локальный файл;

- настроить разный формат логов для консоли и для файла.

При разработке (development), для настройки внешнего вида выводимых в консоль логов, рекомендуют использовать библиотеку [pino-pretty](https://github.com/pinojs/pino-pretty). Однако, использовать библиотеку *pino-pretty* нужно **только во время разработки приложения**., т.е. в ***production*** её использовать можно, но настоятельно ***не рекомендуется***. Библиотека pino-pretty не включена в Fastify по умолчанию из соображений производительности, поэтому её нужно установить с зависимостью **devdependencies**.

```

npm i pino-pretty --save-dev

```

Логи в Pino создаются через механизм "транспорта" (transport). Начиная с **Pino v7 и выше,** транспорты Pino получили возможность работать внутри рабочего потока ([worker threads](https://github.com/GeorgiiGalechyan/nodejs-docs-ru/blob/main/src/markdown/API/Worker%20threads.md)). Для работы с потоками Pino использует библиотеку [thread-stream](https://github.com/pinojs/thread-stream), которая создаёт [stream](https://nodejs.org/api/stream.html) для транспорта. Когда мы создаем поток с помощью [thread-stream](https://github.com/pinojs/thread-stream), [thread-stream](https://github.com/pinojs/thread-stream) порождает рабочий поток ([worker threads](https://github.com/GeorgiiGalechyan/nodejs-docs-ru/blob/main/src/markdown/API/Worker%20threads.md)) (независимый поток выполнения JavaScript-кода).

Для того, чтоб логи одновременно выводились в консоль и сохранялись в файл, нам нужно внутри основного транспортного потока создать 2 (два) независимых рабочих потока, один для работы с консолью, а второй для сохранения в файл.

Для этого в файле ***dev.js*** в объекте конфигурации пропишем следующее:

```javascript

// dev.js

// Создадим массив из двух объектов, каждый из которых является

// конфигурацией отдельного рабочего потока

const targets = [

{

target: 'pino-pretty', // направляет поток выполнения Pino в модуль pino-pretty

options: {

name: 'dev-terminal', // необязательная опция, задает имя потока

level: process.env.PINO\_LOG\_DEV\_LEVEL || 'debug', // уровень логирования

destination: 1, // дефолтное значение, эту строчку можно не указывать

},

},

{

target: 'pino/file', // направляет поток выполнения Pino в файл

options: {

name: 'dev-local-file',

level: 'debug',

destination: './pino/logs/dev.log', // путь к файлу, в который запишутся логи

mkdir: true, // желательно указать когда вы записываете логи в файл

},

},

]

export default {

transport: {

targets,

},

}

```

В указанной конфигурации:

- указав `{ target: 'pino/file' }` мы направили первый поток в модуль pino-pretty, который внутри себя обработал данные из потока и направил в консоль уже изменённые данные;

- указав `{ target: 'pino/file' }` мы указали Pino, что хотим записать данные из второго потока в файл, и задав `{ destination: './pino/logs/dev.log' }` указали Pino путь к локальному файлу, в который следуют записать логи.

На данный момент логирование уже работает. Логи будут выводиться в консоль и сохраняться в файл *./pino/logs/dev.log.* На самом деле, направив первый поток в модуль pino-pretty, мы уже изменили формат сообщения, выводимых регистратором в консоль. Но мы можем их модифицировать в более удобный вид. Pino-pretty предоставляет дополнительные опции для более удобного вывода сообщений в консоль. Мы добавим некоторые, для более тонкой настройки смотрите официальную документацию pino-pretty

Для этого, модифицируем объект конфигурации для первого потока:

```javascript

{

target: 'pino-pretty',

options: {

name: 'dev-terminal',

level: process.env.PINO\_LOG\_DEV\_LEVEL || debug,

// настройки pino-pretty

colorize: true, // добавляем цвета в консоль

customColors: 'err:red,info:green,', // это дефолтные настройки цвета, можно менять

levelFirst: true, // переносим значение level в начало сообщения

ignore: 'pid,hostname', // убираем из сообщения id процесса и имя хоста

include: 'level,time,', // или можно написать то, что мы хотим ставить

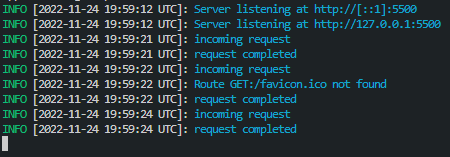
translateTime: 'yyyy-mm-dd HH:MM:ss Z', // меняем формат даты

},

}

```

Полный код файла dev.js можно посмотреть на [gist.github.com](https://gist.github.com/GeorgiiGalechyan/20dc5a4cb951fe526000738f303291ee).

Теперь формат сообщений стал более приятным. В консоли сообщения должны выводиться в таком формате:

А содержание файла ./pino/logs/dev.log выглядит примерно так:

```javascript

**// dev.log**

{"level":30,"time":1669319952479,"pid":2364,"hostname":"MY-PC","msg":"Server listening at http://[::1]:5500"}

{"level":30,"time":1669319952484,"pid":2364,"hostname":"MY-PC","msg":"Server listening at http://127.0.0.1:5500"}

{"level":30,"time":1669319961841,"pid":2364,"hostname":"MY-PC","reqId":"req-1","req":{"method":"GET","url":"/","hostname":"localhost:5500","remoteAddress":"127.0.0.1","remotePort":56116},"msg":"incoming request"}

{"level":30,"time":1669319961854,"pid":2364,"hostname":"MY-PC","reqId":"req-1","res":{"statusCode":200},"responseTime":12.543899983167648,"msg":"request completed"}

{"level":30,"time":1669319962267,"pid":2364,"hostname":"MY-PC","reqId":"req-2","req":{"method":"GET","url":"/favicon.ico","hostname":"localhost:5500","remoteAddress":"127.0.0.1","remotePort":56116},"msg":"incoming request"}

{"level":30,"time":1669319962268,"pid":2364,"hostname":"MY-PC","reqId":"req-2","msg":"Route GET:/favicon.ico not found"}

{"level":30,"time":1669319962269,"pid":2364,"hostname":"MY-PC","reqId":"req-2","res":{"statusCode":404},"responseTime":2.165800005197525,"msg":"request completed"}

{"level":30,"time":1669319964799,"pid":2364,"hostname":"MY-PC","reqId":"req-3","req":{"method":"GET","url":"/","hostname":"localhost:5500","remoteAddress":"127.0.0.1","remotePort":56119},"msg":"incoming request"}

{"level":30,"time":1669319964801,"pid":2364,"hostname":"MY-PC","reqId":"req-3","res":{"statusCode":200},"responseTime":0.988099992275238,"msg":"request completed"}

```

Полный код приложения и текст статьи можно посмотреть в моём GitHub аккаунте.

На этом предлагаю закончить первую часть, а то размер статьи и так превысил мои первоначальные ожидания. Настройку Pino для production предлагаю рассмотрим в следующей статье.

На написание второй части статьи уйдёт какое-то время, т.к. во второй части хочу попробовать направить один из потоков в [Kibana (ELK)](https://www.elastic.co/kibana/) и привести всё это дело в какой-то приличный вид. Я вообще не знаком с Kibana, но мне интересно в нём разобраться и показать вам как Pino интегрируется с внешними инструментами.

Как всегда, буду рад обратной связи, пожеланиям, критике, советам итд. Пишите в комментарии или в личку.