

# A Case Study on the Importance of Custom Logger Keys in Scalable Products

**Do not fear mistakes. You will know failure. Continue to reach out - Benjamin Franklin**

## Introduction

Logging is a very important feature for developers that allows engineers to track data, crackdown on errors, and locate bugs. The open source project that we are working on is called Fastify and it is a web framework for Node.js. Fastify is considered to be one of the most efficient backend frameworks. Not only is logging used to help developers maintain projects, having a universal logging theme is crucial for the transfer of data and services across a wide range of technologies and users.

## Fastify Overview

This open source project is important as the implementation of custom logging keys can enhance efficiency, make products more secure, and help developers debug potential issues in their codebase. Having a well documented and well structured logging system is crucial for the development of a project as it makes troubleshooting incredibly more effective. Having custom logging keys plays a crucial role in scalability and places more emphasis on efficient code. The typical user of the Fastify project is a developer who works with Node.js, which is a server side platform used for web development. Node.js is popular due to its efficient

nature and ability to quickly develop applications. The Fastify framework is useful for web developers as it can help with error handling and has the capability to handle an enormous amount of requests at one time.

```
[nodemon] restarting due to changes...
[nodemon] starting `node server.js`
{"level":30,"time":1731208917464,"pid":6184,"hostname":"SAMRA04","msg":"Server listening at http://127.0.0.1:5000"}
{"level":30,"time":1731208917465,"pid":6184,"hostname":"SAMRA04","msg":"Server listening at http://10.0.0.223:5000"}
{"level":30,"time":1731208932834,"pid":6184,"hostname":"SAMRA04","reqId":"req-1","req":{"method":"GET","url":"/items","host":"localhost:5000","remoteAddress":"10.0.0.223","remotePort":61366},"msg":"incoming request"}
{"level":30,"time":1731208932838,"pid":6184,"hostname":"SAMRA04","reqId":"req-1","res":{"statusCode":200},"responseTime":3.852099895477295,"msg":"request completed"}
```

Fastify Logs in JSON format on my Local Machine (1.1)

According to Statista.com, as of 2024, the Fastify API is currently being used in 2.4% of all web development projects. According to *The Software House*, Fastify is downloaded at least 700,000 times a week and is being used in about 25,000 projects at the time this blog is being posted.

## Exploring the Issue

The issue that we addressed for the project was providing users with the ability to create custom logging keys, similar to how pino-http provides. This implementation of this feature would make it much simpler to integrate the logger with the Google Cloud Structure. Link to issue: <https://github.com/fastify/fastify/issues/4413> This issue is important to the open source project as it implements a feature that already exists with other loggers and has proven to make logging much more developer friendly and cost effective. This issue also would invite more developers to introduce the Fastify framework into their tech stacks as it would make integrating with other technologies seamless. Some of the key locations of the

codebase that are very much relevant to finding the solution to the issue include the fastify.d.ts typescript file which handles server requests and logger options.

```
17  import {
18    FastifyBaseLogger,
19    FastifyChildLoggerFactory,
20    FastifyLogFn,
21    FastifyLoggerInstance,
22    FastifyLoggerOptions,
23    LogLevel,
24    PinoLoggerOptions
25  } from './types/logger'
```

#### A Closer Look at the fastify.d.ts File (2.1)

The next key location within this codebase is the logger.js file which has code imported from pino-http project and this file is responsible for the main functionalities of the Fastify logging system and this is where I have hypothesized that we will implement our custom logger keys. Some other related files that will play a key role in supporting custom attribute keys include the reply.js file and the error.js file as they handle response and error logging. Once custom attributes have been added to the codebase, they will need to be supported by these files in order to provide as much context to developers as possible.

```
51  const serializers = {
52    req: function asReqValue (req) {
53      return {
54        method: req.method,
55        url: req.url,
56        version: req.headers && req.headers['accept-version'],
57        host: req.host,
58        remoteAddress: req.ip,
59        remotePort: req.socket ? req.socket.remotePort : undefined
60      }
61    },
```


#### Definition of Serializers from the 'logger.js' file (2.2)

## Understanding the Inspiration

Although it can be intimidating to begin working on a project of this scale, being able to reference examples such as the Pino-HTTP custom logging keys helps in translating the solution to the issue into executable code. As mentioned in the issue declaration, the inspiration for adding custom attribute keys is derived from

Located within the Pino-HTTP codebase are many intricate details regarding logging keys including, but not limited to custom messages, custom IDs, custom keys, custom props, and custom log levels. In terms of what I believe needs to be implemented into the Fastify logging system, I believe that most important features include custom error messaging, custom IDs, and custom props. Custom error messaging will be crucial for Fastify developers who need access to detailed logs regarding error locations and additional context of when the error took place. Also, custom log IDs will provide developers with the ability to divide their issues into sub categories and assign importance to issues based on their needs. Next, custom props will enhance the logging experience as they will provide developers with the ability to include and/or exclude certain information based on what they view to be the most relevant keys that they need.

```
53 export interface CustomAttributeKeys {  
54     req?: string | undefined;  
55     res?: string | undefined;  
56     err?: string | undefined;  
57     reqId?: string | undefined;  
58     responseTime?: string | undefined;  
59 }
```



Pino-Http Example of Custom Logging Keys (3.1)

# Fastify Codebase Investigation

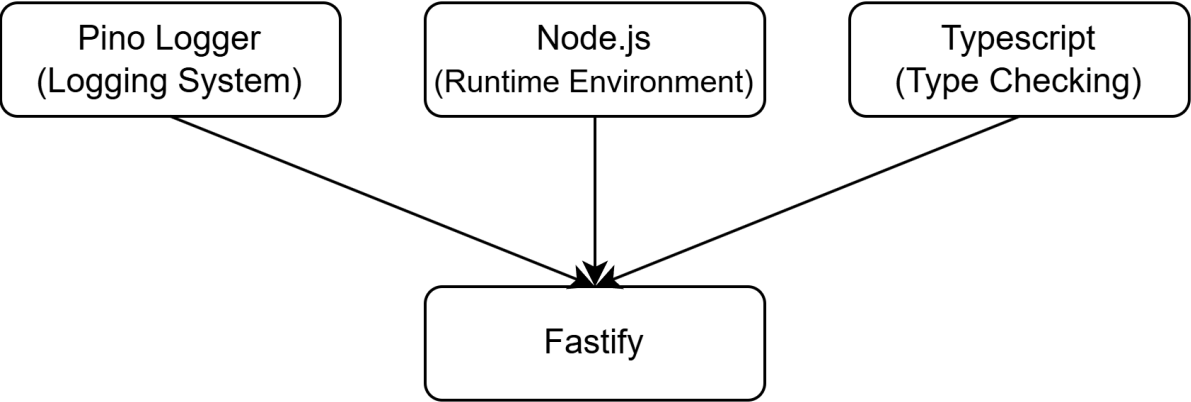
As mentioned in the previous section, the most important file locations in the Fastify codebase consist of 'lib/logger.js', 'fastify.d.ts', 'lib/reply.js', 'lib/error.js'. 'lib/logger' contains the base logic for logger functionality for Fastify. The logger.js file contains the necessary logic for creating loggers as well as allowing developers to create the logger that fits their technical needs (5.1). 'fastify.d.ts' is an important logging support file which utilizes specific TypeScript types in order to support logging functions such as 'request' and 'reply'. This file will be particularly important once custom attribute keys have been implemented as the 'fastify.d.ts' file will have to be updated in order to support the new custom options. Next, 'lib/reply.js' plays a crucial role in sending developers log responses and contains the logic for supporting HTTP responses.

```
59 function Reply (res, request, log) {
60   this.raw = res
61   this[kReplySerializer] = null
62   this[kReplyErrorHandlerCalled] = false
63   this[kReplyIsError] = false
64   this[kReplyIsRunningOnErrorHook] = false
65   this.request = request
66   this[kReplyHeaders] = {}
67   this[kReplyTrailers] = null
68   this[kReplyHasStatusCode] = false
69   this[kReplyStartTime] = undefined
70   this.log = log
71 }
```

Initializing Properties for Logger Response in 'reply.js' (3.2)

Lastly, 'lib/error.js' is an extensive file that contains the error response logic that is present within Fastify, with detailed and custom error responses for most scenarios that a developer may encounter in HTTP request handling.

# The Codebase Skeleton



Fastify Tech Stack (4.1)

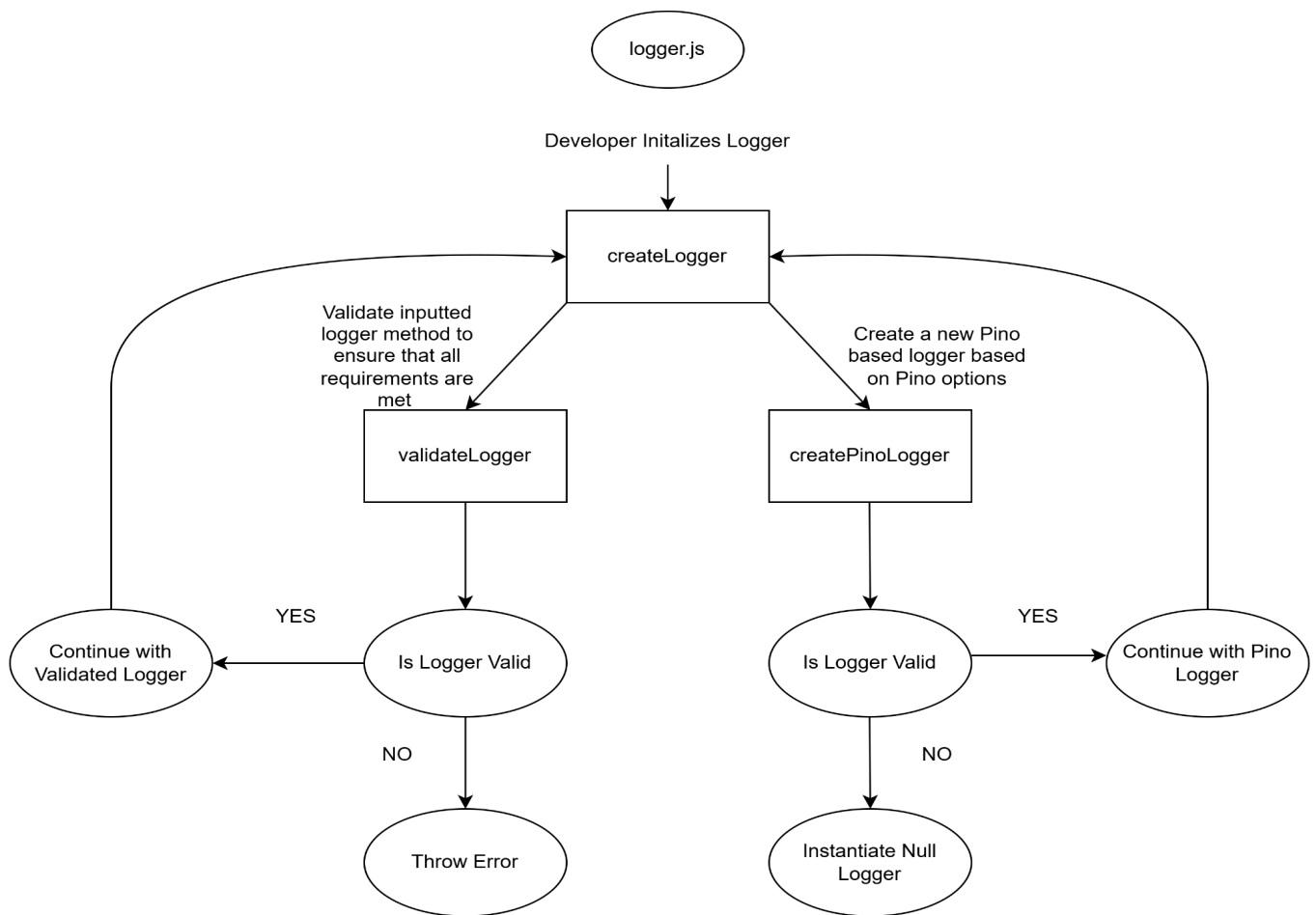
**TECH STACK**

Node.js	Used as the primary base of Fastify and utilizes an event driven model capable of providing efficient and scalable network products.
TypeScript	Typescript is utilized for type checking and for enhancing logger data in order to increase understandability.
Pino Logger	Fastify utilizes the Pino logging system which uses the JSON formatting schema.

## Overcoming Hurdles

A technical challenge that I faced in the beginning of this open source journey was simply being unable to understand the practical importance of logging and why debugging is crucial to the development process. I was able to overcome this challenge fairly quickly by researching first hand accounts of the importance of logging by software engineers who mentioned how they become more productive when they have a standardized way of assessing errors in a codebase. Another specific technical challenge that I have faced during this project is losing track in the codebase. This challenge makes it difficult to trace functions and variables as they operate. Some problem-solving attempts that I have made include making a function roadmap which traces the pathway of the `createLogger` function in the `logger.js` file (5.1). I have also attempted to solve this problem by making predictions of how a specific change in the codebase will operate and testing my hypothesis. If my hypothesis was correct, then I move on to the next phase of the function path, however if I was incorrect, then I form a new hypothesis based on the new information I have. This challenge is still prevalent in terms of my ability to work through the codebase and implement the features that I am trying to add to the project as it is easy to become intimidated by the threat of failure, but I attempt to overcome this feeling by improving my technical skills and my critical thinking on a daily basis.

# Function Roadmaps



logger.js Roadmap (5.1)

## And Then There Were Two

After conducting my research and exploring both the Fastify codebase and the Pino-HTTP codebase, I concluded that there are two potential ways forward from here:

- Implement `customAttributeKeys` within Fastify.
- Implement `customAttributeKeys` within Pino.



Each solution contains its own pros and cons as I will dissect now. Implementing `customAttributeKeys` within Fastify would likely be the more preferable option for the Fastify maintainers as it would incorporate custom attribute keys directly inside of Fastify instead of needing to synchronize a complex feature with another codebase. Also, implementing this feature directly inside Fastify would allow Fastify developers to have streamlined access to an important logging feature that would help them dramatically improve their efficiency. The major con of implementing this feature inside Fastify is that it would increase the complexity of the logging system and potentially make debugging harder for a developer. On the other hand, implementing `customAttributeKeys` in Pino would allow Fastify to maintain its efficient nature and low overhead capabilities while boosting debugging productivity. Plus, implementing `customAttributeKeys` within Pino would provide a more universal feature that could be used in other applications instead of just Fastify and it would keep the Fastify codebase simpler, meaning that the learning curve for the Fastify codebase would not become steeper. The negative aspect of this route is that it would require developers to familiarize themselves with the Pino logging system to a greater degree and the actual implementation of custom attribute keys would require the developer to have a solid understanding in the logical networks utilized in the Pino codebase.

## Summary

The implementation of custom attribute keys is a necessary utility that would drastically improve the organizational structure of the Fastify framework and the best solution to this issue would be to implement the custom attribute keys within the Pino codebase. By doing so, Fastify would retain its efficient response handling,

high performance, and low overhead while providing a universal addition to the Pino logging system. This is the path that I will take as I continue to work through this issue, familiarize myself with the importance of efficient logging standards, and help the entire Fastify developer community by enhancing their ability to debug and customize logging in a scalable, effective manner.