UNIVERSIDADE ESTADUAL DA PARAÍBA
CAMPUS VII - GOVERNADOR ANTÔNIO MARIZ
CENTRO DE CIÊNCIAS EXATAS E SOCIAIS APLICADAS
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

LUCAS HENRIQUE OLIVEIRA DE ARAUJO

EXPRESS E FASTIFY: UM ESTUDO DE COMPARAÇÃO ENTRE DOIS FRAMEWORKS NODE.JS

PATOS
2024

LUCAS HENRIQUE OLIVEIRA DE ARAUJO

EXPRESS E FASTIFY: UM ESTUDO DE COMPARAÇÃO ENTRE DOIS FRAMEWORKS WEB NODE.JS

Trabalho de Conclusão de Curso apresentado ao Curso de Bacharelado em Ciência da Computação da Universidade Estadual da Paraíba, em cumprimento à exigência para obtenção do grau de Bacharel em Ciência da Computação.

Área de concentração: Desenvolvimento Web
Orientador: Profa. Giovanna Trigueiro de Almeida Araujo

PATOS
2024

A663e Araújo, Lucas Henrique Oliveira de.

Express e *fastify* [manuscrito] : um estudo de comparação entre dois *frameworks web node.js* / Lucas Henrique Oliveira de Araújo. - 2024.

52 f.

Digitado.

Trabalho de Conclusão de Curso (Graduação em Ciência da computação) - Universidade Estadual da Paraíba, Centro de Ciências Exatas e Sociais Aplicadas, 2024.

"Orientação : Prof. Esp. Giovanna Trigueiro de Almeida Araújo, Coordenação do Curso de Computação - CCEA".

1. Desenvolvimento web. 2. Node.js. 3. Benchmarking. 4. Frameworks back-end. I. Título

21. ed. CDD 006.7

Elaborada por Kaliane Eveny Martins de Oliveira - CRB - 15/986 BSC7

LUCAS HENRIQUE OLIVEIRA DE ARAUJO

EXPRESS E FASTIFY: UM ESTUDO DE COMPARAÇÃO ENTRE DOIS FRAMEWORKS WEB NODE.JS

Trabalho de Conclusão de Curso apresentado à Coordenação do Curso de Ciência da Computação da Universidade Estadual da Paraíba, como requisito parcial à obtenção do título de Bacharel em Ciência da Computação

Aprovada em: 22/11/2024.

Documento assinado eletronicamente por:

- **Harllem Alves do Nascimento** (***.796.924-**), em **01/12/2024 18:34:34** com chave **0ea43d60b02c11efa86f06adb0a3afce**.
- **Rodrigo Alves Costa** (***.667.224-**), em **01/12/2024 14:00:27** com chave **c385a9cab00511efab9e06adb0a3afce**.
- **Giovanna Trigueiro de Almeida Araújo** (***.352.004-**), em **01/12/2024 11:25:04** com chave **0eb83946aff011efa57a06adb0a3afce**.

Documento emitido pelo SUAP. Para comprovar sua autenticidade, faça a leitura do QrCode ao lado ou acesse https://suap.uepb.edu.br/comum/autenticar_documento/ e informe os dados a seguir.
Tipo de Documento: Termo de Aprovação de Projeto Final
Data da Emissão: 02/12/2024
Código de Autenticação: 054144

Dedico este trabalho à minha querida mãe Sônia Maria (in memoriam), que cuidou de mim até seus últimos dias e sempre acreditou no meu potencial, me ensinando a ser forte e a transmitir o amor e bondade.

AGRADECIMENTOS

Ao meu pai, Raimundo, que é meu principal exemplo de vida e sempre me apoiou e incentivou em todos os momentos da minha jornada de formação.

Ao meu irmão, Eduardo, que foi desde o início minha principal motivação para iniciar minha formação na área e me fez querer ser constantemente melhor naquilo que faço.

À minha querida namorada Millena, que é a mais pura e vívida representação da paz, carinho e amor, e que esteve do meu lado durante toda a produção deste trabalho. Meu amor e gratidão por você não cabem nessas palavras.

Às pequenas Helena, Valentina, Alice e Cecília, que me trazem tantas alegrias e aliviaram diversos momentos durante todo o meu percurso de formação.

Aos meus amigos, em especial, Lucas, Andreano, Adriano e Nicolas, que se fizeram presentes durante grande parte da minha trajetória acadêmica.

À professora Giovanna, que sempre se mostrou prestativa, me auxiliando e orientando durante todo o processo de construção deste trabalho.

Aos funcionários e professores da UEPB, em especial Pablo Suarez, Rodrigo Costa e Rosângela, que contribuíram diretamente para o meu aprendizado e na construção do conhecimento.

Aos meus colegas de turma, que por muitas vezes foram responsáveis por me transmitir novos conhecimentos e tornar o ambiente de aprendizado em um lugar mais leve.

Por fim, agradeço a todas as pessoas que fizeram parte e contribuíram para a minha formação. A todos muito obrigado.

RESUMO

No contexto atual de constante evolução dos processos de desenvolvimento web, a escolha do *framework* back-end para o desenvolvimento de APIs é fundamental para o sucesso de um projeto. Este fator é muito importante, já que há uma crescente demanda no mercado por aplicações performáticas e escaláveis. A partir dessa perspectiva, este estudo visa comparar de forma quantitativa e qualitativa dois *frameworks* Node.js amplamente utilizados no desenvolvimento de APIs REST: Express e Fastify. Para realizar a comparação, uma mesma aplicação foi implementada utilizando ambos *frameworks*, sendo testadas por meio de uma ferramenta de *benchmarking*, obtendo dados de latência, taxa de requisições por segundo e de leitura de dados. Além disso, para a coleta de dados para análise qualitativa, foram utilizados aspectos de popularidade de cada *framework*. A análise dos resultados indicou que o Fastify apresenta maior eficiência nos parâmetros dos testes, sendo uma opção mais performática, ideal para aplicações que exigem alta eficiência. Já o Express, apesar de inferior nos aspectos de *performance*, mantém-se relevante devido à sua popularidade, documentação e facilidade de uso, sendo útil em projetos que priorizam simplicidade, suporte e familiaridade. Esses dados contribuem para decisões mais informadas sobre a escolha de *frameworks* no desenvolvimento de APIs em Node.js, oferecendo *insights* para otimização de desempenho e adequação tecnológica conforme as necessidades do projeto.

Palavras-Chave: desenvolvimento web; Node.js; *benchmarking*; *frameworks back-end*.

ABSTRACT

In the current context of the constant evolution of web development processes, the choice of a back-end framework for API development is essential for a project's success. This factor is crucial, as there is a growing market demand for high-performance and scalable applications. From this perspective, this study aims to quantitatively and qualitatively compare two widely used Node.js frameworks for REST API development: Express and Fastify. To conduct the comparison, the same application was implemented using both frameworks and tested through a benchmarking tool, gathering data on latency, requests per second, and data read rate. Additionally, to collect data for qualitative analysis, aspects of each framework's popularity were considered. The results indicated that Fastify offers greater efficiency in the test parameters, making it a more high-performance option ideal for applications requiring high efficiency. Express, while less efficient in performance aspects, remains relevant due to its popularity, documentation, and ease of use, proving useful in projects that prioritize simplicity, support, and familiarity. This data contributes to more informed decisions about framework selection for API development in Node.js, providing insights for performance optimization and technological alignment according to project needs.

Keywords: web development; Node.js; *benchmarking*; *back-end frameworks*.

LISTA DE FIGURAS

LISTA DE ABREVIATURAS E SIGLAS

API	Application Programming Interface
ARP	API de Relatórios de Ping
E/S	Entrada/Saída
HTTP	HyperText Transfer Protocol
HTTPS	HyperText Transfer Protocol Secure
ORM	Object-Relational Mapping
REST	Representational State Transfer
URI	Uniform Resource Identifier
URL	Uniform Resource Locator

SUMÁRIO

1 INTRODUÇÃO

O desenvolvimento de APIs REST é algo essencial no ambiente de desenvolvimento web, pois supre a crescente demanda mercadológica por aplicações altamente performáticas e escaláveis. Nesse contexto, *frameworks* Node.js, como Express e Fastify, conseguiram uma notoriedade considerável devido a sua flexibilidade, simplicidade e eficiência. Porém, ainda existe uma lacuna de conhecimento no meio acadêmico, principalmente na língua lusófona, em relação à comparação de performance entre esses *frameworks*, deixando desenvolvedores e organizações incertos sobre a escolha da ferramenta ideal para maximizar a eficiência de suas aplicações.

A relevância de uma comparação entre Express e Fastify está em identificar qual framework possui a melhor capacidade de performance no processamento de requisições HTTP. A escolha do framework ideal no contexto do desenvolvimento de uma API influencia de maneira significativa a eficiência operacional da aplicação, afetando a experiência do usuário e a escalabilidade da aplicação. Em um ambiente onde o desempenho é crucial, tal análise comparativa fornece informações valiosas na tomada de decisões técnicas fundamentadas.

Este estudo tem como objetivo realizar uma avaliação detalhada de performance entre os *frameworks* Fastify e Express, utilizando uma abordagem quantitativa baseada em testes de *benchmark*. Por meio da implementação de uma mesma aplicação REST em ambos os *frameworks*, seguida da realização de testes de desempenho, será possível realizar a medição e comparação de métricas como quantidade de requisições processadas em um determinado intervalo de tempo, latência e taxa de dados.

Os resultados dessa análise têm como objetivo contribuir de forma relevante para a literatura técnica e prática do desenvolvimento de APIs REST no ambiente Node.js, apresentando informações válidas que possam auxiliar aqueles envolvidos no desenvolvimento de aplicações web, na escolha da ferramenta mais adequada para o seu contexto e necessidades específicas. Por fim, é esperado que este estudo evidencie as diferenças de performance entre Express e Fastify e impulsione a adoção de práticas mais eficientes no desenvolvimento de aplicações web backend em Node.js.

1.1 Problematização

Apesar de os *frameworks* web baseados em Node.js, Express e Fastify terem conseguido uma grande notoriedade no mercado de desenvolvimento de APIs REST, ainda existe uma falta de entendimento na maneira em como essas ferramentas se comparam em relação à eficiência no processamento de requisições HTTP. Essa lacuna de conhecimento, acaba gerando uma dúvida em relação à qual das duas ferramentas é a mais apropriada dentro dos diversos contextos possíveis no desenvolvimento de uma aplicação web back-end de alta performance. Esse fato se torna relevante em um cenário onde a demanda por aplicações performáticas e escaláveis cresce constantemente.

A partir disso, nasce a questão que esse trabalho se propõe a responder: com a realização de uma análise comparativa de performance entre os *frameworks* Express e Fastify, qual é a ferramenta ideal para a construção de uma API REST de alta performance, levando em consideração o contexto do seu desenvolvimento?

1.2 Objetivos

1.2.1 Objetivo geral

Analisar a eficiência em termos de processamento de requisições HTTP dos *frameworks* Express e Fastify, através da medição da latência, taxa de dados processados e quantidade de requisições processadas por uma aplicação em um determinado intervalo de tempo, buscando determinar a diferença de performance entre as duas ferramentas.

1.2.2 Objetivos específicos

- Desenvolver a mesma aplicação para ambas ferramentas;
- Realizar testes de performance nas aplicações desenvolvidas;
- Coletar e analisar os dados dos testes;

1.3 Justificativa

A escolha da ferramenta ideal para o desenvolvimento de aplicações web back-end em Node.js é de extrema importância para garantir a eficiência e o desempenho do sistema. Nesse contexto, Express e Fastify, surgem como duas ferramentas muito populares na construção de APIs REST, porém ainda há uma certa ausência de produção no meio acadêmico, principalmente na produção de artigos em língua portuguesa, em relação à eficiência de cada um desses *frameworks*.

Nesse cenário, torna-se relevante a realização de uma análise comparativa detalhada das diferenças de performance de aplicações utilizando ambas ferramentas por meio de testes de desempenho.

1.4 Procedimentos metodológicos

O estudo proposto foi orientado através de uma abordagem mista, utilizando elementos quantitativos e qualitativos, sendo composto por três fases principais: desenvolvimento, testes de performance e análise de dados. Na primeira fase foi desenvolvida uma mesma aplicação em ambos *frameworks* estudados, que contou com o conjunto de funcionalidades mais presentes em APIs REST web: criação, listagem, atualização e exclusão de dados. Na fase seguinte, foram realizados os testes em um ambiente controlado a fim de garantir a confiabilidade dos resultados. O ambiente foi configurado em uma máquina com especificações de hardware e software claramente definidas para evitar variações no desempenho devido a fatores externos.

A principal ferramenta para realização dos testes de performance foi o AutoCannon, que é uma biblioteca do ambiente Node.js, o que facilita a sua utilização e configuração, já que os objetos de estudo têm como base esse mesmo ambiente. Além disso, a ferramenta já foi utilizada em dois estudos que fundamentam parte da base teórica desta pesquisa, Demashov (2023) e Wikander (2020), com um propósito muito próximo ao da sua utilização neste trabalho: coletar de forma prática, métricas precisas relativas à performance no contexto da comparação da eficiência de tecnologias web. O AutoCannon possui a capacidade

de enviar múltiplas requisições simultâneas, permitindo também determinar a quantidade de conexões simultâneas e o método HTTP utilizado, viabilizando dessa forma uma avaliação mais precisa dos desempenhos das principais operações das aplicações desenvolvidas em Express e Fastify.

Na terceira e última fase, ocorreu a coleta e análise dos dados de performance, que incluiu métricas como a latência, tempo de resposta mínimo, máximo, médio e por percentil, taxa de dados e requisições por segundos. A análise foi realizada a partir da comparação das métricas de desempenho de ambos *frameworks*, utilizando o método comparativo, que segundo Fachin (2005), viabiliza a análise de dados concretos e a partir daí, a dedução de elementos gerais, constantes e abstratos. No contexto deste estudo, esse método permitiu determinar as diferenças entre os *frameworks* avaliados, com base nas métricas coletadas.

2 FUNDAMENTAÇÃO TEÓRICA

Esta seção aborda de forma aprofundada os principais conceitos, fundamentos e tecnologias envolvidas, fornecendo uma visão mais clara sobre o contexto em que esse trabalho está inserido. O intuito é construir um embasamento teórico forte, dando sustentação para a realização da análise comparativa entre os *frameworks* Express e Fastify, através da execução de testes de performance.

A princípio, na subseção 2.1, serão mostrados os principais aspectos do Node.js, apresentando suas características principais, além do aumento da sua popularidade no mercado no desenvolvimento de aplicações web modernas. Na subseção 2.2, será explorado o conceito de API REST, que se trata de uma das principais utilizações dos *frameworks* Express e Fastify, detalhando seu funcionamento e relevância na criação de aplicações web back-end. Por fim, a subseção 2.3, tratará das principais tecnologias utilizadas para a realização dessa análise comparativa.

2.1 Node.js

O Node.js (comumente chamado de Node), conforme Node (2024), é definido como um ambiente de execução JavaScript multiplataforma e de código aberto que executa o motor JavaScript V8, que é o núcleo do Google Chrome, fora do navegador, permitindo que o Node.js seja muito performático.

O Node.js foi criado por Ryan Dahl em 2009, tendo como principal objetivo oferecer uma maneira eficiente de lidar com aplicações baseadas em rede, utilizando o modelo de E/S não bloqueante. Segundo Dahl (2018), a inspiração para o projeto veio da necessidade de criar servidores que pudessem lidar com múltiplas conexões simultâneas de maneira eficiente.

Após a primeira release feita em 2009, o Node.js começou a receber mais atenção no meio do desenvolvimento JavaScript, recebendo em 2010, de acordo com Hámori (2024), um suporte experimental na plataforma de aplicações em nuvem, Heroku. Ainda de acordo com Hámori (2024), o gerenciador de pacotes do Node, ou simplesmente npm, recebeu sua *release* 1.0 em maio de 2011,

alavancando ainda mais a popularidade da plataforma, que começou a ser utilizada na rede social LinkedIn em agosto do mesmo ano.

Depois desse ganho de popularidade, ainda segundo Hámori (2024), o Node.js começou a receber diversas contribuições da comunidade de desenvolvedores JavaScript, principalmente com a implementação de diversos novos *frameworks*, além de começar a ser utilizado em ambiente de produção em diversos outros projetos de empresas de grande porte, como o aplicativo móvel Uber, no site de compra e venda de produtos, eBay e na plataforma de streaming de vídeo da Netflix, além de diversas outras empresas que começaram a utilizar o Node em seus mais variados projetos.

De acordo com Demashov (2023), o Node.js ganha destaque no contexto da construção de APIs REST, pois apresenta um tratamento assíncrono de solicitações baseado em um sistema de geração de eventos, onde cada solicitação gera um evento que é encaminhado para uma fila de processamento, com isso o Node completa a resposta para a solicitação quando há tempo de execução. Esse padrão de execução do Node impede o uso exagerado de memória, ao mesmo tempo em que evita o bloqueio de processo.

De acordo com Node (2024), o modelo de geração de eventos apresentado anteriormente é conhecido no Node.js como loop de eventos sendo o responsável por permitir a execução de operações de entrada e saída de maneira não bloqueante. Quando o Node.js executa uma operação de E/S, como por exemplo o acesso a um banco de dados, ao invés de bloquear a *thread* perdendo ciclos de CPU enquanto aguarda, o Node dará continuidade às operações com a chegada da resposta. Desse modo, o Node.js consegue lidar com milhares de conexões simultâneas em um único servidor, sem precisar gerenciar a concorrência de *threads*, tornando-se uma alternativa plausível no desenvolvimento de sistemas escaláveis, como por exemplo APIs REST.

Para Huang (2020) a utilização do sistema de E/S não bloqueantes, implementado utilizando a arquitetura orientada a eventos, pode ser considerada uma das características mais marcantes do Node.js. Além disso, o modelo de arquitetura utilizada no sistema não bloqueante de processos do Node, de acordo com Jansen (2024), permite uma maior resiliência do sistema de eventos

implementado, uma vez que os componentes geradores de eventos não necessitam saber sobre o status de saúde dos componentes consumidores e vice-versa.

Além de um sistema de processamento de requisições robusto e bem estruturado, com a utilização de um modelo de loop de eventos baseado em uma arquitetura escalável e performática, o Node.js:

> [...] tem uma vantagem única porque milhões de desenvolvedores front-end que escrevem JavaScript para o navegador agora podem escrever o código do lado do servidor, além do código do lado do cliente, sem a necessidade de aprender uma linguagem completamente diferente (Node, 2024, tradução própria).

A popularidade no Node.js no mercado de desenvolvimento web facilitou o surgimento de um vasto ecossistema, que atingiu em setembro de 2022 o número de 2,1 milhões pacotes (incluindo bibliotecas e *frameworks* como Express e Fastify) registrados no NPM, fazendo com que o gerenciador de pacotes do Node se tornasse o maior repositório de código de uma única linguagem no planeta, como aponta Node (2024).

Esse grande número de bibliotecas e *frameworks* registrados no gerenciador de pacotes no Node é justificado da seguinte maneira:

> [...] a plataforma Node.js é uma ferramenta suficiente para desenvolver sistemas REST, mas surge um problema mais especializado ao criar aplicações: a escolha de um framework apropriado. Essa questão apareceu porque, originalmente, quando a plataforma começou sua evolução, não havia um grande framework adequado para implementar aplicações de qualquer complexidade (Demashov, 2023, tradução própria).

Demashov (2023), destaca que como resultado dessa questão, os desenvolvedores começaram, de maneira independente, a implementar suas próprias soluções, tomando como base os módulos padrões do Node.js e como consequência disso o ecossistema do Node apresenta atualmente muitos *frameworks* e bibliotecas que tem como foco a implementação de sistemas web.

Graças à estrutura de funcionamento do Node.js que utiliza um design baseado em um modelo assíncrono e orientado a eventos, é possível chegar a conclusão de que o Node.js se destaca em diversos aspectos, possuindo diversas possíveis aplicações técnicas como no desenvolvimento de aplicações que

funcionam com base em comunicação em tempo real, como por exemplo chats de texto, que possivelmente necessitam gerenciar múltiplas conexões simultâneas.

Outra possível utilização, já destacada anteriormente por Demashov (2023), é a criação de APIs REST. Além disso, devido a sua leveza e modularidade o Node também facilita a construção de aplicações com arquiteturas baseadas em microsserviços, permitindo escalabilidade horizontal com facilidade. Além dessas aplicações, o Node também beneficia aplicações que lidam com grande volume de dados pois permite o processamento de dados durante o upload ou download, ao invés de aguardar a conclusão completa da transferência

2.2 APIs REST

Di Meglio (2019) define REST como um tipo de arquitetura cliente/servidor que fornece simplicidade e escalabilidade, além de facilitar a interoperabilidade entre diversos sistemas que usam o protocolo HTTP em uma rede para se comunicar. O REST se baseia no conceito de recursos, onde cada recurso, além de possuir um URI único para identificação, é um objeto que precisa ser consultado ou manipulado através da rede.

Ainda segundo Di Meglio (2019), uma API REST expõe os seus recursos, permitindo que os clientes realizem diversas operações como: buscar, apagar ou editar recursos existentes e criar novos recursos. O autor evidencia que:

> As operações são tipicamente realizadas por meio de requisições HTTP para os URIs apropriados, usando métodos padrão como GET, POST, PUT e DELETE, que se alinham semanticamente com as ações básicas de recuperar, criar, atualizar e deletar recursos, respectivamente (Di Meglio, 2019, tradução própria).

Relan (2019), destaca que a principal característica da arquitetura REST é a sua ausência de estado (*stateless*), isso permite que o cliente não precise conhecer o estado do servidor e vice-versa, desse modo, ambos podem entender todas mensagens recebidas sem necessitar saber qual foi a mensagem anterior. O fato de o REST ser *stateless* torna sua implementação simples em qualquer modelo de sistema, além de possibilitar que sistemas diferentes respondam requisições diferentes.

Outras características da arquitetura REST, citadas por Zhou et al. (2014), são as seguintes: o REST não utiliza registro centralizado, dependendo de conexões entre recursos para gerenciar e descobrir novos recursos; o REST possui clientes heterogêneos, permitindo ajustes na representação de recursos e protocolos, baseado nas capacidades de cada cliente e nas condições da rede; esse modelo de arquitetura também suporta composições orientadas a serviços, independentemente da linguagem de programação ou plataforma; a escalabilidade também uma característica muito importante, e deve pelo fato de que o REST é *stateless*, melhorando o desempenho através de caches em camadas.

Em seu estudo, Di Meglio (2019) destaca que:

> Devido à sua simplicidade, escalabilidade e interface uniforme e padronizada, o estilo arquitetural REST emergiu como uma escolha predominante na indústria de software para desenvolver serviços web robustos e interoperáveis [...] Atualmente, uma ampla gama de *frameworks* de API REST está disponível, escritos em diferentes linguagens de programação (Di Meglio, 2019, tradução própria).

Os *frameworks* focados na construção de APIs REST, disponibilizam para os desenvolvedores diversos recursos, como bibliotecas, funções, conceitos e abstrações pré-construídas para facilitar e simplificar a construção, gerenciamento e manutenção de APIs que aplicam os diversos conceitos da arquitetura REST.

2.3 Tecnologias escolhidas

2.3.1 Express

De acordo com Express (2024), o Express é um framework Node.js de código aberto, minimalista e flexível, que fornece um conjunto robusto de funcionalidades e recursos para desenvolvimento de aplicações web. Seu principal foco está no desenvolvimento de APIs, pois fornece uma grande quantidade de métodos utilitários HTTP e middlewares.

Segundo Express (2024), a primeira release do *framework* data de março de 2010, um momento em que o Node surgia como uma tecnologia promissora. O Express foi criado por TJ Holowaychuk e rapidamente se tornou o *framework* mais

popular da plataforma do Node, visto que na época devido o Node.js ainda era uma tecnologia muito recente e não possuía tantos pacotes ou bibliotecas. Segundo Humand (2024), em 2014 Holowaychuk vendeu os direitos do projeto para a StrongLoop, uma empresa que atua especialmente com Node.js, logo em seguida, a IBM adquiriu os direitos do Express. Por fim, em janeiro de 2016, a IBM colocou o Express sob a liderança da Node.js Foundation.

Mozilla (2024), afirma que o Express é o *framework* Node.js mais popular, como mostrou a pesquisa realizada pelo Stack Overflow em 2023, onde o Express apareceu com sendo utilizado por cerca de 20% dos desenvolvedores profissionais e também por aqueles que estão aprendendo. De acordo com Humand (2024), essa popularidade se deve por conta de características como o minimalismo, a versatilidade e a acessibilidade do ExpressJS.

Ainda segundo Mozilla (2024) o Express, oferece diversas soluções para criação de aplicações web, como por exemplo: o gerenciamento de requisições dos vários métodos HTTP em URLs diferentes, a adição de novos processos para uma requisição, em qualquer ponto da fila requisições, através da implementação de *middlewares* e também a definição de configurações da aplicação, como a porta de conexão, localização das estruturas usadas para renderizar a resposta.

Além disso, o Express não define nenhum tipo de modelo de organização de pastas ou nenhum tipo de estrutura a ser utilizada para renderização de uma resposta, como Express (2024) afirma:

> Para ser o mais flexível possível, o Express não faz suposições em termos de estrutura. Rotas e outras lógicas específicas do aplicativo podem residir em quantos arquivos você desejar, em qualquer estrutura de diretório de sua preferência (Express, 2024, tradução própria).

Segundo Mardan (2014), sistemas desenvolvidos em Express permitem que os desenvolvedores escolham livremente as bibliotecas do ambiente Node.js para implementar funcionalidades, portanto, aplicações Express.js são altamente configuráveis.

2.3.2 Fastify

Fastify (2024) afirma que a tecnologia Fastify é um *framework* web que possui uma poderosa arquitetura de plugins, além de ser focado na melhoria da experiência do desenvolvedor, possuindo o mínimo de sobrecarga possível, sendo considerado um dos *frameworks* web mais rápidos da atualidade.

O Fastify, de acordo com Fastify (2024) tem a sua primeira release datada em outubro de 2016, sendo criado pelo desenvolvedor *open source* Matteo Collina, com o objetivo de fornecer uma alternativa mais rápida e eficiente aos frameworks Node.js já existentes. Atualmente diversas organizações e empresas utilizam o Fastify em seus projetos de APIs, algumas delas estão listadas no site oficial do *framework*, como por exemplo, NearFrom, Microsoft e Mercedes.

Assim como o Express.js o Fastify também é uma ferramenta muito popular no desenvolvimento de APIs e gerenciamento de back-end, sendo utilizado por 9% dos mais de 20 mil participantes da pesquisa State of JavaScript (2022), que ocorre anualmente, onde é realizada a coleta de dados a partir da resposta dos desenvolvedores à um questionário completo sobre a linguagem JavaScript e as tecnologias que se baseiam nela, com o objetivo de obter de informações sobre o atual estado das tecnologias e linguagem, e suas comunidades.

De acordo com o portal Rocketseat (2024), o Fastify é utilizado principalmente no desenvolvimento de APIs REST:

> Graças à sua performance superior e ao suporte integrado para serialização e validação de JSON, o Fastify é ideal para desenvolver APIs que necessitam de alta velocidade de resposta e eficiência no tratamento de dados. [...] O Fastify foi projetado com o foco na performance, utilizando técnicas como o parsing eficiente de JSON e a otimização de rotas para acelerar o processamento de requisições (Rocketseat, 2024).

O Fastify, como pode ser notado em Fastify (2024), apresenta mais de 60 plugins principais, desenvolvidos pela equipe principal do projeto, ainda mais importante, também apresenta uma comunidade de desenvolvedores muito ativa e comprometida com a manutenção do *framework*, sendo grande responsável por desenvolver e compor um ecossistema rico e continuamente otimizado, com mais de 200 plugins desenvolvidos por essa rede comunitária de programadores.

2.3.3 JavaScript

De acordo com Mozilla (2024), JavaScript é uma linguagem interpretada e baseada em objetos, sendo a mais utilizada nos scripts de páginas web, assim como em outros ambientes fora do navegador como o Node.js. Mozilla ainda afirma que: “O JavaScript é uma linguagem baseada em protótipos, multi-paradigma e dinâmica, suportando estilos de orientação a objetos, imperativos e declarativos (como por exemplo a programação funcional)”.

De acordo com GeeksForGeeks (2024), o JavaScript foi desenvolvido por Brendan Eich num período de tempo de apenas 10 dias em 1995, sendo utilizado como a linguagem de script do navegador Netscape 2, que era o mais popular na época, tendo sido lançado em 1996. Após isso, a Microsoft rapidamente adotou o JavaScript como linguagem para o seu navegador Internet Explorer. A facilidade de uso da linguagem e posição única como a única linguagem de script do lado do cliente tornaram o JavaScript popular entre os desenvolvedores web, sendo hoje em dia uma das linguagens mais utilizadas no mundo.

Segundo Stack Overflow (2023), sua pesquisa anual realizada entre os dias 8 e 19 de maio de 2023, com mais de 80 mil respostas, mostrou que o JavaScript é a linguagem mais popular na comunidade de desenvolvedores por onze anos seguidos, com mais de 60% de utilização por parte dos participantes da pesquisa. A popularidade do JavaScript se dá pelo fato de:

> [...] ele ser uma linguagem que suporta tanto o paradigma de programação orientada a objetos quanto o paradigma funcional. Essa versatilidade permite que os desenvolvedores utilizem a linguagem em uma ampla variedade de contextos, tornando-a uma escolha preferencial em muitos cenários de desenvolvimento (De Araujo, 2023 apud Ryder et. al., 2015).

2.3.4 TypeScript

Segundo TypeScript (2024), o Typescript é uma linguagem de programação fortemente tipada baseada em JavaScript, e que fornece ferramentas que ajudam o desenvolvedor a escalar projetos mais facilmente, por meio de funcionalidades como a definição de tipos de variáveis personalizados e a realização da tipagem de dados

por inferência. De acordo com Invedus (2024), o TypeScript foi criado dentro da Microsoft, pelo engenheiro de software Anders Hejlsberg, que iniciou o desenvolvimento da linguagem em 2010, tendo a primeira release pública em 2012.

É válido destacar também que o TypeScript:

> [...] enriquece JavaScript com um sistema de módulos, classes, interfaces e um sistema de tipos estáticos. Como TypeScript visa fornecer assistência leve aos programadores, o sistema de módulos e o sistema de tipos são flexíveis e fáceis de usar. Em particular, eles suportam muitas práticas comuns de programação JavaScript (Bierman et. al., 2014, tradução própria).

2.3.5 Autocannon

O AutoCannon foi criado por Matteo Collina em abril de 2016 e é descrito por Collina (2024), como sendo uma ferramenta de benchmarking HTTP/1.1 escrita em Node, já tendo sido utilizado como ferramenta de benchmarking nos trabalhos de Demashov (2023) e Wikander (2020), ambos trabalhos compõem parte da fundamentação teórica desta pesquisa.

Existem diversas ferramentas similares que cumprem o mesmo propósito de benchmarking HTTP, como por exemplo o Bombardier e o Apache JMeter, apesar disso, o AutoCannon foi selecionado para este trabalho por possuir uma documentação mais ampla e clara quando comparado ao Bombardier, além do fato de que é uma ferramenta escrita em Node.js, diferentemente do Apache JMeter.

O fato de o AutoCannon ser escrito em Node.js reforça ainda mais sua escolha para esta pesquisa uma vez que ele é projetado especificamente para testar servidores HTTP criados com essa tecnologia, garantindo uma integração perfeita e eliminando possíveis gargalos causados por incompatibilidades de ferramentas externas não nativas do ecossistema Node.js, podendo resultar em diferenças na simulação do tráfego típico de uma aplicação Node.

Além disso, o AutoCannon é muito popular dentro da comunidade de desenvolvedor que utilizam Node.js, sendo amplamente utilizados em testes de *benchmarking* na comunidade, garantindo atualizações regulares e suporte para desenvolvedores que precisam de confiabilidade e compatibilidade com novas versões do Node.js

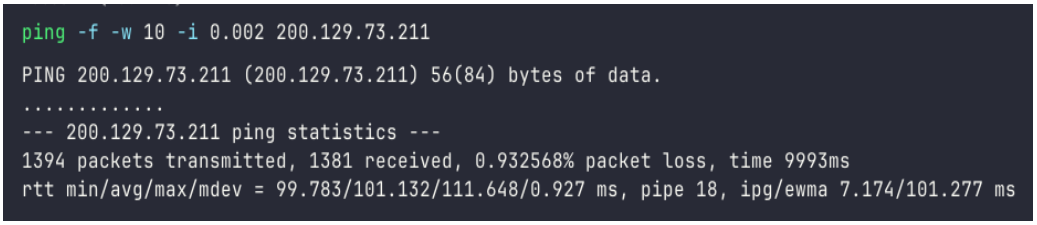
3 ASPECTOS DE IMPLEMENTAÇÃO

3.1 API de Relatórios de Ping

A API de Relatórios de Ping ou simplesmente ARP, foi a aplicação criada para essa pesquisa. Essa aplicação armazena relatórios de *ping* obtidos a partir da saída da execução de um comando *ping* para um IP específico, sendo possível criar, editar e remover relatórios, assim como listar todos os relatórios cadastrados, a partir de requisições HTTP.

Cada relatório de *ping* apresenta uma estrutura simples, baseada na saída do comando *ping* que é comumente usado para testar a conectividade entre dois dispositivos em uma rede, e apresenta em sua saída os dados de quantidade de pacotes enviados e recebidos, a taxa de perda, o tempo total de execução e os tempos de ida e volta mínimo, médio e máximo, como pode ser observado na captura de tela abaixo:

Figura 1 - Execução de comando *ping*



Fonte: Capturado pelo autor, 2024

A ARP foi replicada utilizando ambos os *frameworks* em foco neste trabalho, sendo importante destacar as similaridades das implementações. Começando na forma como os dados foram modelados para o banco de dados, que se deu por conta da utilização de uma ORM chamada Prisma, muito utilizada em aplicações Node.js. O Prisma, segundo Prisma (2024), é uma ORM de próxima geração que pode ser utilizada em qualquer aplicação *backend* Node.js e também com utilização e integração com TypeScript. A utilização dessa ORM também levou a escolha do PostgreSQL como banco de dados das aplicações, já que ele é o banco de dados padrão do Prisma, o que facilita a configuração inicial e oferece uma integração

rápida e eficiente com a aplicação, tornando o desenvolvimento mais ágil e prático. É possível observar o modelo dos dados utilizando Prisma na imagem abaixo:

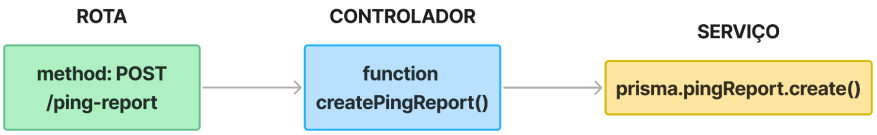
Figura 2 - Modelo de relatório de ping via Prisma ORM

Fonte: Elaborado pelo autor, 2024

Além da utilização de uma mesma ORM, ambas implementações da ARP possuem uma mesma estrutura, que é comumente utilizada em aplicações de *frameworks* baseados em Node.js, contando com um conjunto de rotas, controladores e serviços. Para cada chamada realizada para uma rota encarregada por uma operação da API, uma função controladora é acionada, sendo responsável pela sequência de funcionamento daquela operação. A partir da execução da função controladora, os serviços necessários para a realização da operação são enfim acionados, com a finalidade de realizar diretamente a manipulação dos dados no banco de dados (nesse caso o acesso ao banco e a manipulação do mesmo ocorre a partir de uma instância da Prisma ORM).

A imagem abaixo descreve de forma simplificada o funcionamento da estrutura descrita acima, a partir de um exemplo de chamada do método POST para uma rota de criação de relatórios de *ping*:

Figura 3 - Diagrama da estrutura da aplicação



Fonte: Elaborado pelo autor, 2024

As rotas da ARP possuem uma estrutura de escrita bastante similar entre cada implementação da aplicação, apresentando apenas algumas pequenas diferenças de sintaxe entre cada *framework*. Porém, as rotas funcionam de maneira idêntica, de modo que, cada rota atende a um método HTTP específico e é responsável por acionar as funções controladoras que processam os dados ou parâmetros da requisição. As declarações das rotas da aplicação desenvolvida com Express podem ser observadas na imagem abaixo, onde as rotas de criação, listagem, edição e deleção dos relatórios de *ping* estão sendo implementadas:

Figura 4 - Declaração das rotas com Express

```
import { Router } from "express"
import { pingReportController } from "./ping.controller"

const router = Router()

router.route("/").get(pingReportController.findAll)
router.route("/").post(pingReportController.create)
router.route("/").put(pingReportController.update)
router.route("/").delete(pingReportController.remove)

export default router
```

Fonte: Elaborado pelo autor, 2024

As declarações das rotas da aplicação desenvolvida com Fastify podem ser vistas na figura abaixo, onde as rotas de criação, listagem, edição e deleção dos relatórios de *ping* estão sendo criadas:

Figura 5 - Declaração das rotas com Fastify

```
import { FastifyInstance } from "fastify"
import { pingReportController } from "./ping.controller"

export const router = async (server: FastifyInstance) => {
    server.get("/", pingReportController.findAll)
    server.post("/", pingReportController.create)
    server.put("/", pingReportController.update)
    server.delete("/", pingReportController.remove)
}
```

Fonte: Elaborado pelo autor, 2024

A partir da chamada de uma rota, a função controladora associada a ela é executada. De forma breve, a função controladora recebe os dados necessários a partir do corpo ou do *endpoint* da requisição, dependendo do método associado àquela rota. É importante destacar que em ambas implementações as funções controladoras possuem a mesma lógica de funcionamento para cada método específico, ou seja, se uma função criada na implementação da ARP com Express é acionada por uma rota de método POST e possui uma lógica, a função criada na implementação com Fastify que atende àquele método, também possui a mesma sequência lógica.

De modo geral, as sequências lógicas estabelecidas para cada função controladora seguem um mesmo método de funcionamento, começando com um tratamento de erros associados à problemas de conexão, seguido por uma verificação ou validação dos dados recebidos (dependendo da finalidade da função) e no final há uma chamada de um serviço gerado pela instância da Prisma ORM, que é responsável pelo acesso ao banco de dados.

As funções associadas à edição e deleção de objetos, possuem uma validação de erros responsável pela verificação da presença do objeto a ser modificado no banco de dados, por meio do identificador único daquele objeto. Vale destacar que nas funções de edição e inserção de dados, há uma validação das informações recebidas, para garantir que os dados possuem o formato correto. As funções de listagem possuem apenas uma verificação de erros de conexão e a partir disso executam o serviço responsável pela listagem de todos os relatórios.

Nas imagens abaixo é possível observar os modelos genéricos de cada tipo de função, associadas às operações de criação, listagem, edição e deleção de dados, utilizados em ambas as implementações. Todas as funções mostradas a seguir apresentam os mesmos parâmetros de corpo de requisição e de resposta, exceto pela função de listagem, que utiliza apenas o corpo de resposta.

A imagem seguinte apresenta a implementação de um modelo de função responsável pela criação de relatórios. A função valida o corpo da requisição por meio de um tipo que descreve a forma de como a entrada de dados deve ser passada para a criação. Após isso ocorre a execução do serviço Prisma encarregado de realizar a inserção dos dados no banco de dados, depois disso é enviada uma resposta de sucesso.

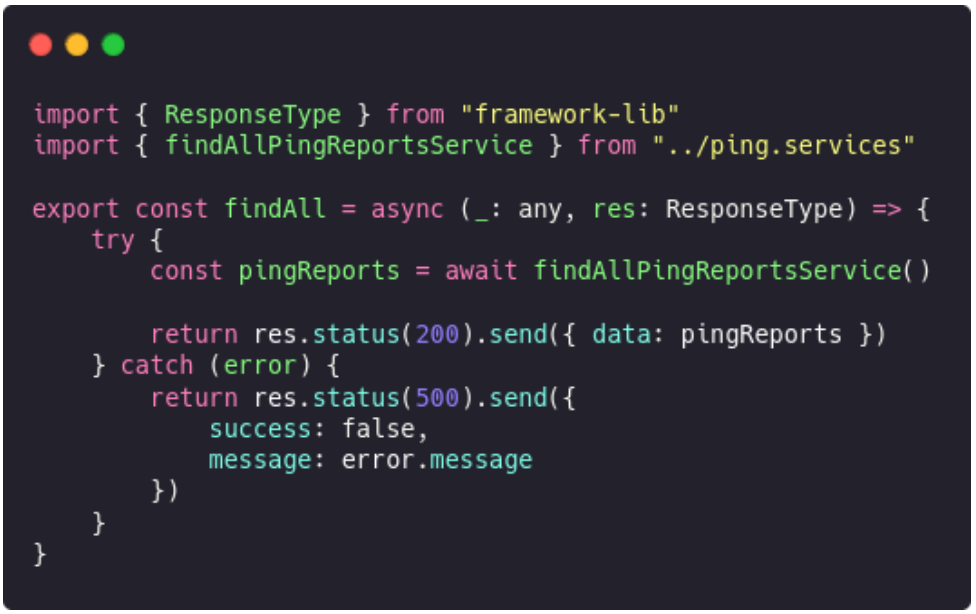
Figura 6 - Modelo genérico de função controladora de criação de relatórios

```
import { RequestType, ResponseType } from "framework-lib"
import { createPingReportService } from "../ping.services"

export const create = async (req: RequestType, res: ResponseType) => {
    try {
        const body: PingReportCreateInput = req.body
        const pingReportData = await createPingReportService(body)

        return res.status(201).send({ success: true })
    } catch (error) {
        return res.status(500).send({
            success: false,
            message: error.message
        })
    }
}
```

Fonte: Elaborado pelo autor, 2024

A imagem a seguir mostra a declaração de um modelo genérico de função controladora que lista relatórios de *ping*. A função utiliza um serviço Prisma encarregado de realizar a busca dos dados no banco de dados, depois disso é enviada uma resposta de sucesso, junto aos dados listados.

Figura 7 - Modelo genérico de função controladora de listagem de relatórios

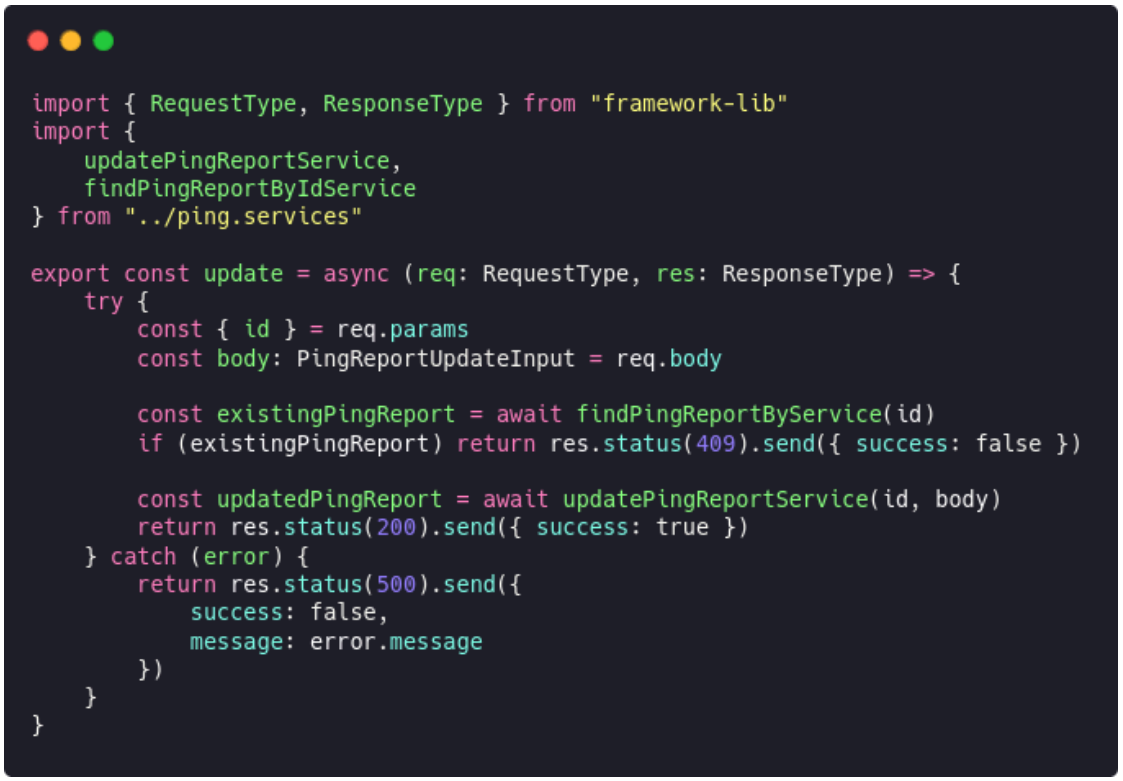


```
import { ResponseType } from "framework-lib"
import { findAllPingReportsService } from "../ping.services"

export const findAll = async (_: any, res: ResponseType) => {
    try {
        const pingReports = await findAllPingReportsService()

        return res.status(200).send({ data: pingReports })
    } catch (error) {
        return res.status(500).send({
            success: false,
            message: error.message
        })
    }
}
```

Fonte: Elaborado pelo autor, 2024

A imagem seguinte apresenta a implementação de um modelo de função responsável pela edição de relatórios. A função valida o corpo da requisição por meio de um tipo que descreve a forma de como a entrada de dados deve ser passada para a edição, além de verificar a existência do relatório a ser editado no banco de dados, por meio de uma chamada de serviço Prisma para busca por identificador único. Depois ocorre a execução do serviço Prisma encarregado da edição dos dados no banco de dados e uma resposta de sucesso é enviada.

Figura 8 - Modelo genérico de função controladora de edição de relatórios

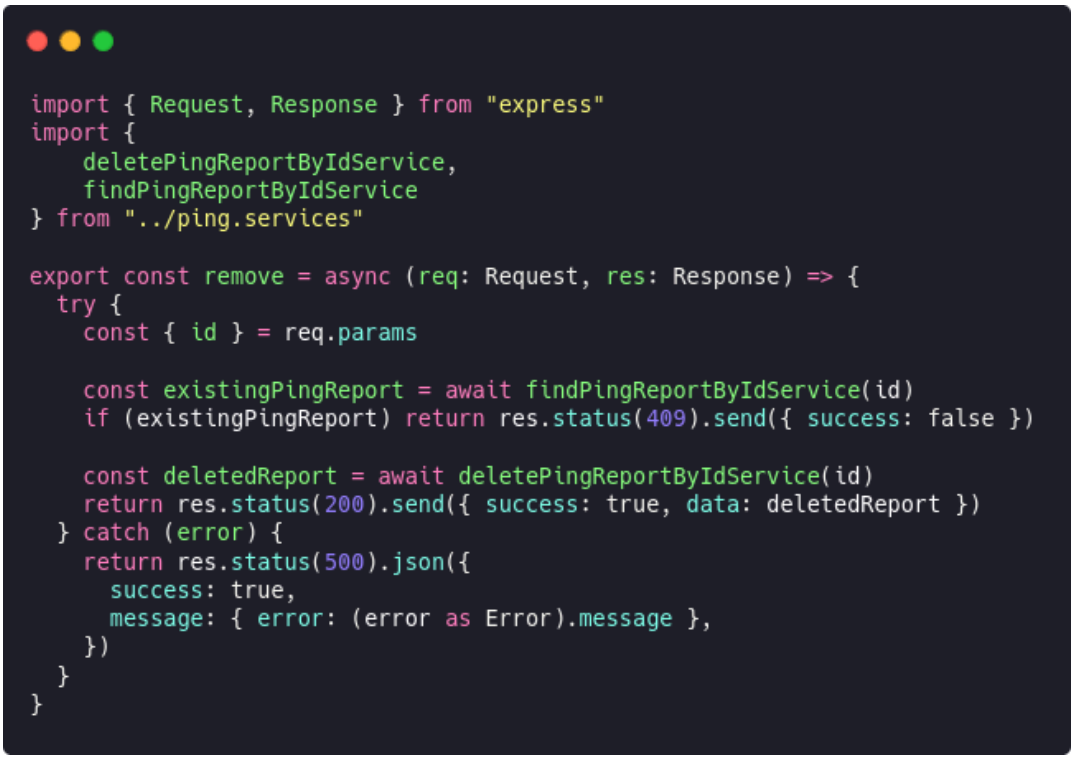


```
import { RequestType, ResponseType } from "framework-lib"
import {
    updatePingReportService,
    findPingReportByIdService
} from "../ping.services"

export const update = async (req: RequestType, res: ResponseType) => {
    try {
        const { id } = req.params
        const body: PingReportUpdateInput = req.body

        const existingPingReport = await findPingReportByService(id)
        if (existingPingReport) return res.status(409).send({ success: false })

        const updatedPingReport = await updatePingReportService(id, body)
        return res.status(200).send({ success: true })
    } catch (error) {
        return res.status(500).send({
            success: false,
            message: error.message
        })
    }
}
```

Fonte: Elaborado pelo autor, 2024

A imagem abaixo mostra a estrutura de um modelo de função que apaga relatórios. A função verifica a existência do relatório a ser editado no banco de dados, por meio de uma chamada de serviço Prisma para busca por identificador único. Depois ocorre a execução do serviço Prisma encarregado da edição dos dados no banco de dados e uma resposta de sucesso é enviada.

Figura 9 - Modelo genérico de função controladora de deleção de relatórios



```
import { Request, Response } from "express"
import {
    deletePingReportByIdService,
    findPingReportByIdService
} from "../ping.services"

export const remove = async (req: Request, res: Response) => {
  try {
    const { id } = req.params

    const existingPingReport = await findPingReportByIdService(id)
    if (existingPingReport) return res.status(409).send({ success: false })

    const deletedReport = await deletePingReportByIdService(id)
    return res.status(200).send({ success: true, data: deletedReport })
  } catch (error) {
    return res.status(500).json({
      success: true,
      message: { error: (error as Error).message },
    })
  }
}
```

Fonte: Elaborado pelo autor, 2024

Além dessas funções controladoras associadas às rotas das operações da ARP, cada implementação também contou com uma rota de checagem de status de saúde da API, que indica por meio de um retorno de um valor booleano se a aplicação está sendo executada corretamente. Essa rota não possui uma função controladora, mas também foi utilizada para os testes para registrar as métricas de quantidade total de requisições de forma mais precisa.

4 RESULTADOS

A seguir será mostrada a forma como foram executados os testes de performance e o processo de coleta dos dados, além de uma análise dos dados obtidos e de uma análise qualitativa. Serão detalhados os parâmetros configurados para a execução dos testes, assim como as ferramentas e abordagens utilizadas.

4.1 Execução dos testes e processo de coleta

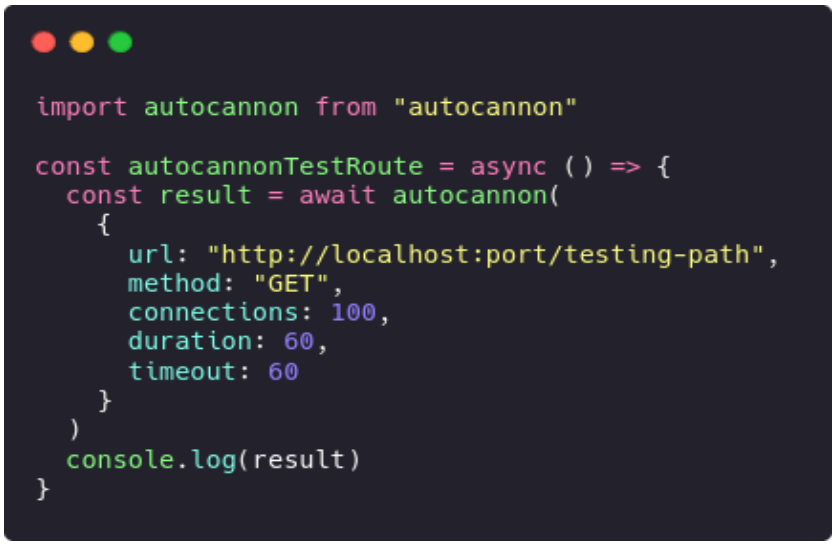
Para a realização dos testes de desempenho e coleta dos dados, foi utilizada uma biblioteca do ambiente Node chamada AutoCannon. Essa biblioteca tem a capacidade de enviar diversas requisições em um curto espaço de tempo, podendo gerar também uma determinada quantidade de conexões simultâneas, que pode ser definida pelo usuário. A biblioteca permite importar uma função para a execução dos testes, essa função possui diversos argumentos, porém os mais utilizados ao longo dos testes foram aqueles destinados a especificar a quantidade de conexões simultâneas, duração do teste, método da rota e tempo máximo de espera.

Além desses, em rotas em que é necessário o envio de dados no corpo ou no caminho da requisição (como nas rotas relacionadas com a criação, edição e deleção), se fez necessário utilizar os parâmetros que determinam o corpo ou o caminho específico, por exemplo, as funções relacionadas às rotas de deleção precisam de um identificador único do objeto a ser deletado, que é informado através do caminho da rota.

Em todos os testes a quantidade de conexões simultâneas foi especificada em 100 e a duração dos testes foi definida em 60 segundos. Para os testes das rotas de listagem, atualização e deleção foi determinado o tempo máximo de espera de resposta das requisições de 60 segundos, para evitar que os testes dessas rotas fossem concluídos com requisições incompletas, o que poderia gerar uma inconsistência nos dados.

Um modelo genérico de uma função utilizando o AutoCannon para a realização de um teste em uma rota específica, pode ser observado na imagem abaixo:

Figura 10 - Modelo genérico de uma função utilizando o AutoCannon para a realização de um teste



```
import autocannon from "autocannon"

const autocannonTestRoute = async () => {
  const result = await autocannon(
    {
      url: "http://localhost:port/testing-path",
      method: "GET",
      connections: 100,
      duration: 60,
      timeout: 60
    }
  )
  console.log(result)
}
```

Fonte: Elaborado pelo autor, 2024

Os dados dos testes foram coletados a partir da saída da execução dos testes com AutoCannon. A coleta ocorreu de forma manual, a partir da observação dos valores apresentados nas saídas de cada testes e após isso os dados foram inseridos em uma planilha do Google Planilhas (aplicação de planilhas do Google, amplamente utilizada para a geração de gráficos, que é o caso de uso da pesquisa, além de análise de dados e cálculos específicos que podem variar de acordo com o contexto) onde ocorreu a geração de gráficos de barra, com o objetivo de facilitar a análise dos resultados utilizando uma representação gráfica.

Para os testes de listagem, o banco de dados foi populado com 1000 linhas de dados de relatório.

Todos os testes foram realizados em uma máquina física executando o sistema operacional Ubuntu na sua versão 24.04. As especificações de *hardware* são: CPU Intel i5-3330 3200 Ghz, GPU AMD ATI Radeon RX 580, além de possuir 16GB de memória RAM e armazenamento de 500GB.

Nas próximas seções serão mostradas as representações gráficas dos testes para cada rota. Para os testes de rotas que utilizam o método GET (método HTTP geralmente usado para recuperar dados de um servidor), no caso as rotas de listagem e verificação de status, foram colhidos os dados referentes à latência, requisições por segundo e taxa de leitura de dados. Nas rotas de verificação de status a principal observação a ser feita será em relação à quantidade de

requisições totais realizadas, uma vez que o tamanho da resposta é pequena, permitindo que o AutoCannon realize uma quantidade muito maior de requisições durante o período dos testes.

Já para os testes das rotas que utilizam os métodos POST (método HTTP usado para criar novos recursos no servidor), PUT (método HTTP utilizado para atualizar ou substituir um recurso existente) e DELETE (método HTTP usado para excluir um recurso no servidor), nesse caso correspondendo respectivamente as rotas de criação, edição e deleção, foram colhidos apenas os dados de latência e requisições por segundo, uma vez que, para essas rotas a taxa de leitura de dados não é uma informação relevante para a medição de performance, já que a utilização desses métodos (principalmente devido à forma como foram implementadas as aplicações) não envolve a leitura direta de grandes volumes de dados.

Os resultados dos testes de latência utilizando AutoCannon são apresentados em termos de percentis e de uma média geral dos tempos de resposta das requisições. Os percentis indicam a distribuição dos dados durante o teste, sendo eles: 2,5%, 50%, 97,5% e 99%. Esses valores correspondem, respectivamente, aos tempos de resposta mais rápidos, à mediana, aos tempos de resposta mais lentos e aos tempos de resposta extremamente lentos. Desse modo, o percentil de 2,5% representa os valores abaixo dos 2,5% do total de requisições que foram atendidas, o percentil de 50%, representa os valores abaixo dos 50% e assim por diante. Esse modelo de indicação é válido para todos os percentis de todos os testes.

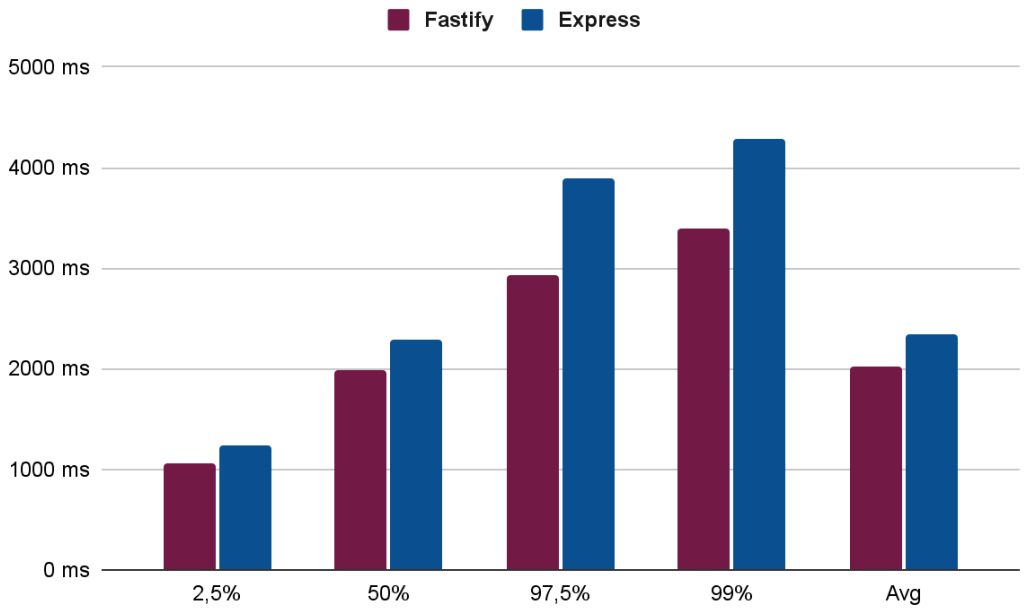
Assim como nos testes de latência, os resultados dos testes de quantidade de requisições por segundo e taxa de leitura de dados são apresentados em termos de percentis, porém há uma mudança nos valores dos percentis, que passam a ser 1%, 2,5%, 50% e 97,5%, e além da média geral, também são apresentados os valores de desvio padrão e o valor mínimo registrado durante o teste.

A análise dos resultados será realizada a partir da observação de cada conjunto de percentis, principalmente para os que fornecerem uma visão e interpretação mais clara dos desempenhos dos *frameworks*.

4.2 Rotas de listagem: latência, requisições por segundo e leitura de dados

Os resultados dos testes realizados nas rotas de listagem de cada implementação utilizando Express e Fastify podem ser observados nas figuras abaixo. Esta seção contém a análise desses resultados, nos permitindo compreender a diferença de desempenho relativa a cada aspecto apresentado.

Figura 11 - Representação gráfica dos testes de rotas de listagem referentes à latência

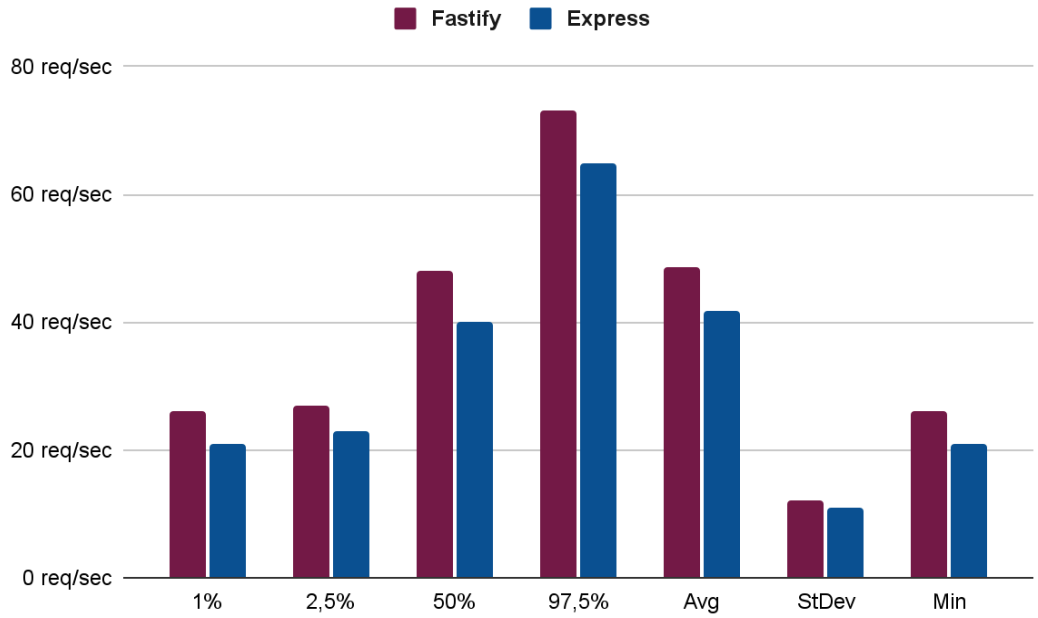


Fonte: Elaborado pelo autor, 2024

A figura 11 nos mostra os resultados relacionados à latência das rotas de listagem de cada *framework*, sendo possível observar uma diferença entre os tempos de resposta de cada aplicação na indicação de todos os percentis, principalmente nos percentis de 97,5% e 99%. Essa diferença indica que a aplicação implementada com Fastify apresentou tempos de resposta mais rápidos, relação ao Express, para a maioria das requisições.

A discrepância também pode ser notada a partir da observação dos valores registrados nas médias finais, onde o Express apresentou uma latência média de 2343,88 milissegundos e o Fastify de 2022,36 milissegundos. Aplicando o cálculo de diferença percentual entre dois valores (que consiste na divisão da diferença dos valores analisados pelo valor de referência, e após isso, a multiplicação desse resultado por 100) nas médias apresentadas, é possível notar que o Express foi aproximadamente 15,9% mais lento que o Fastify.

Figura 12 - Representação gráfica dos testes de rotas de listagem referentes à requisições por segundo

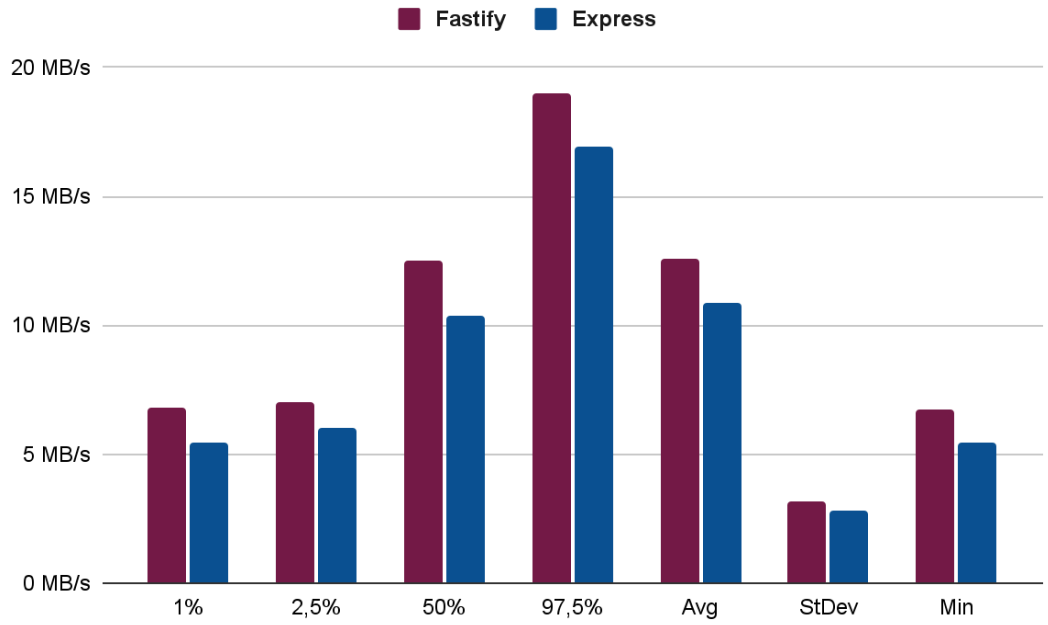


Fonte: Elaborado pelo autor, 2024

A partir da observação da figura 12, que indica os resultados relativos à quantidade de requisições por segundo das rotas de listagem, é possível notar que em todas as indicações, tanto de percentis e quanto de valores mínimos, há uma diferença constante, entre as aplicações, onde a aplicação implementada com Fastify conseguiu executar mais requisições no intervalo de tempo de um segundo que a aplicação desenvolvida com Express.

Observando os valores da média, percebe-se que o Fastify fez em média 48,6 requisições por segundo e o Express 41,84. Dessa forma, verifica-se que através do cálculo da diferença dos valores das médias que o Fastify realizou aproximadamente 6,76 requisições por segundo a mais que o Express. Além disso, o desvio padrão registrado em ambos os testes foi bastante similar, o que mostra a pouca variação dos valores de cada percentil em relação à média geral registrada em ambas as aplicações.

Figura 13 - Representação gráfica dos testes de rotas de listagem referentes à taxa de leitura de dados



Fonte: Elaborado pelo autor, 2024

Seguindo com a análise dos dados dos testes, a figura 13 foi obtida a partir da coleta dos dados relativos à taxa de leitura de dados. A partir da observação do gráfico mostrado na figura conclui-se que o Fastify obteve uma taxa de leitura maior que o Express em todas as medidas de percentis, com destaque para os valores de 97,5% e 99%, assim como no valor mínimo, onde a disparidade foi de aproximadamente 1,29 MB/s.

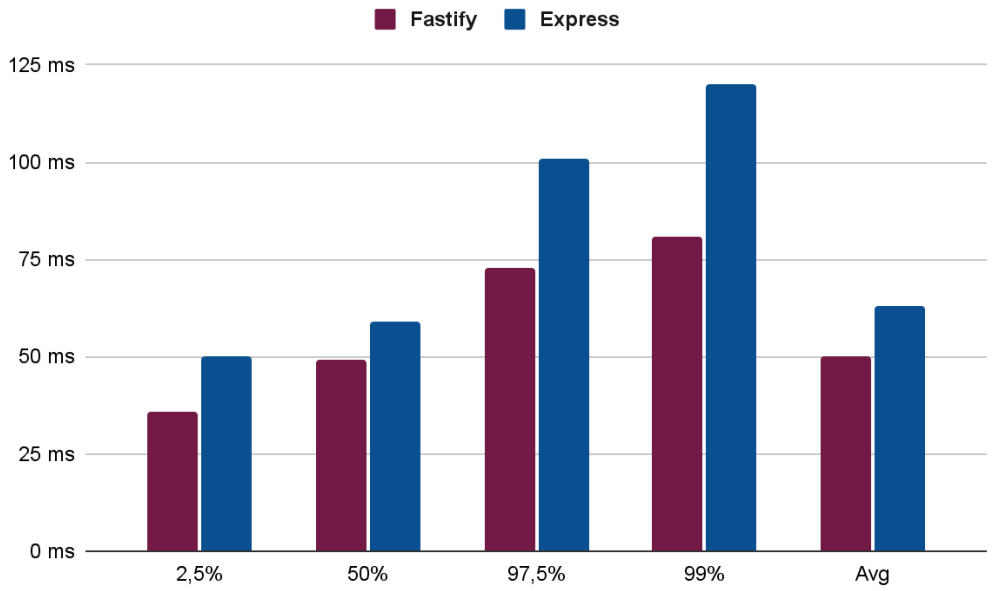
A diferença constante entre o desempenho de leitura de dados se refletiu nos valores da média, onde o Fastify apresentou uma taxa de leitura de dados de aproximadamente 15,6% maior em comparação ao Express. O desvio padrão apresentado foi relativamente baixo, o que indica uma variação baixa dos valores em relação à média registrada.

4.3 Rotas de criação: latência e requisições por segundo

Esta seção contém a análise dos resultados dos testes realizados nas rotas de criação para cada aplicação desenvolvida com os *frameworks* Express e Fastify. Os dados de latência e o número de requisições por segundo, apresentados nas

figuras a seguir, fornecem uma visão mais detalhada sobre os diferentes desempenhos dos *frameworks* na operação de criação de dados.

Figura 14 - Representação gráfica dos testes de rotas de criação referentes à latência

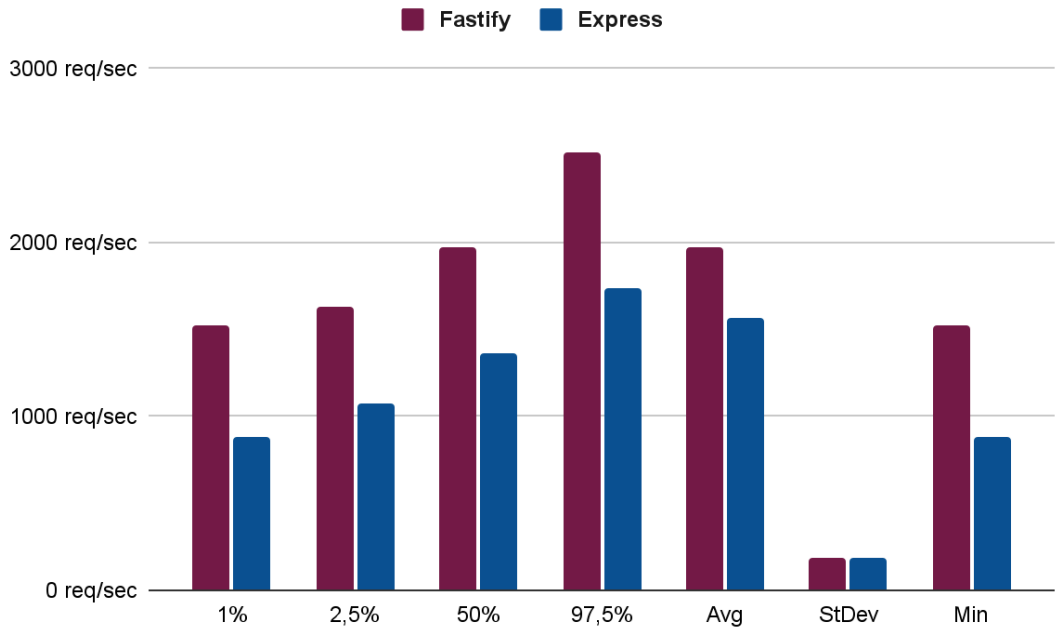


Fonte: Elaborado pelo autor, 2024

A figura acima apresenta os resultados dos testes de latência das rotas de criação. A diferença entre os tempos de resposta é evidente em todos os valores de percentis, sendo possível notar uma grande discrepância entre os valores dos percentis de 97,5% e no de 99%, onde a disparidade chega a ser de 39 milissegundos. A menor desigualdade dos resultados pode ser observada nos percentis de 50%, sendo de apenas 10 milissegundos.

Os valores das médias finais das implementações com Express e Fastify, correspondem respectivamente a 63,24 milissegundos e a 50,15 milissegundos. Desse modo, assim como foi feito para os resultados das rotas de listagem, aplicando o cálculo da variação percentual, constata-se que o Express se mostrou, na média, aproximadamente 26,1% mais lento que o Fastify.

Figura 15 - Representação gráfica dos testes de rotas de criação referentes à requisições por segundo



Fonte: Elaborado pelo autor, 2024

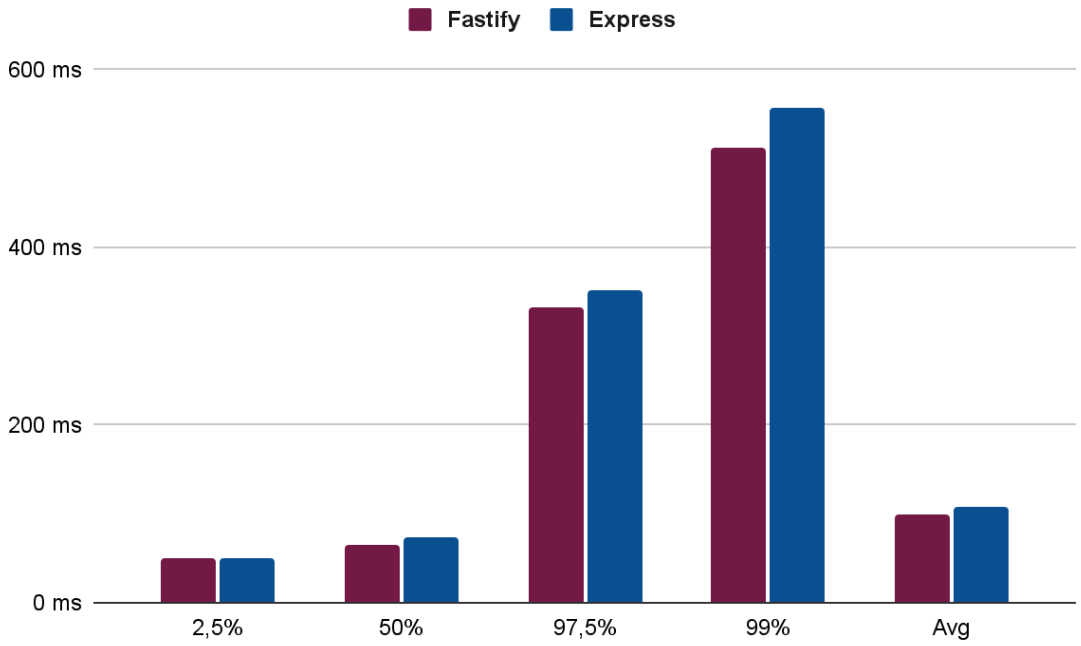
A partir da observação da figura 15, é possível notar uma grande disparidade de valores em todas as indicações de percentis. Destacando apenas as amostras dos termos de 1%, percebe-se que a aplicação construída com Fastify realizou 647 requisições por segundo a mais que a implementada com Express. Na marca de 97,5% a discrepância é ainda maior, correspondendo a 789 requisições a mais por parte da aplicação com Fastify em comparação com a aplicação que utiliza Express.

Além disso, constata-se os baixos valores de desvio padrão para ambos *frameworks*, o que se reflete nos resultados de modo que para grande parte deles não há uma variação tão grande em relação a média geral, que foi de aproximadamente 1568,9 requisições por segundo para a implementação com Express e de 1975,5 para a implementação com Fastify. Observando a diferença desses valores, o Fastify fez em torno de 406,54 requisições por segundo a mais que o Express, no mesmo período de tempo.

4.4 Rotas de edição: latência e requisições por segundo

Esta seção apresenta a avaliação dos resultados dos testes das rotas de edição de cada aplicação desenvolvida com Express e Fastify. Os resultados analisados a seguir são relativos aos parâmetros de latência e número de requisições por segundo, essa análise fornecerá uma visão mais clara dos desempenhos dos *frameworks* na operação de edição de dados.

Figura 16 - Representação gráfica dos testes de rotas de edição referentes à latência

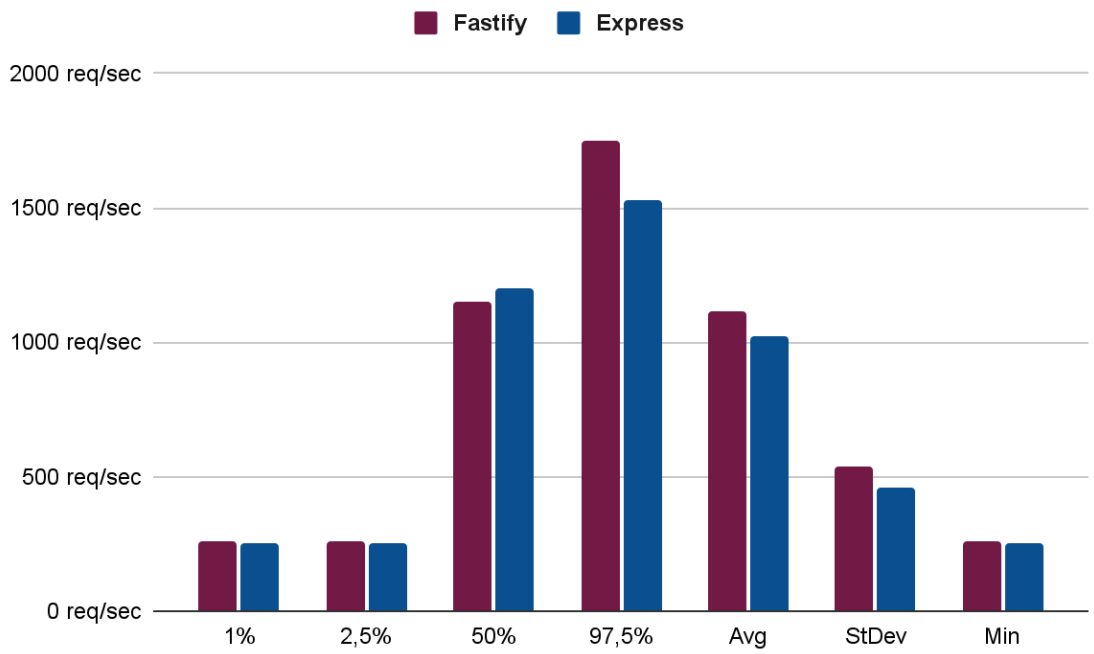


Fonte: Elaborado pelo autor, 2024

A figura 16, mostrada acima, fornece uma perspectiva de que há um equilíbrio de desempenho entre os dois *frameworks* na execução da operação de edição de dados, com Fastify apresentando tempos um pouco menores que o Express. Pode-se perceber que os valores dos percentis de 2,5%, 50% e 97,5% são muito similares: no percentil de 2,5%, o Fastify e o Express apresentaram o mesmo valor de latência, 51 milissegundos; no percentil de 50%, o Fastify apresentou 65 milissegundos, enquanto o Express, 74 milissegundos; e no percentil de 97,5%, o Fastify registrou 332 milissegundos, enquanto o Express 351 milissegundos.

A maior diferença pode ser observada no percentil de 99%, onde o Fastify foi de 46 milissegundos mais rápido que o Express. O equilíbrio dos valores se refletiu nos valores das médias finais, onde a partir da aplicação do cálculo de diferença percentual, pode-se constatar que o Fastify foi aproximadamente 8,7% mais rápido que o Express.

Figura 17 - Representação gráfica dos testes de rotas de edição referentes à requisições por segundo



Fonte: Elaborado pelo autor, 2024

Observando a figura 17, verifica-se de forma clara um equilíbrio dos valores para a maioria dos percentis, como pode ser notado nos percentis de 1%, 2,5% e 50%, este último mostrando uma leve vantagem 51 de requisições por segundo do Express sobre o Fastify. O percentil de 97,5% foge um pouco do claro equilíbrio observado anteriormente, pois o Fastify apresentou uma vantagem de cerca de 200 requisições por segundo em relação ao Express.

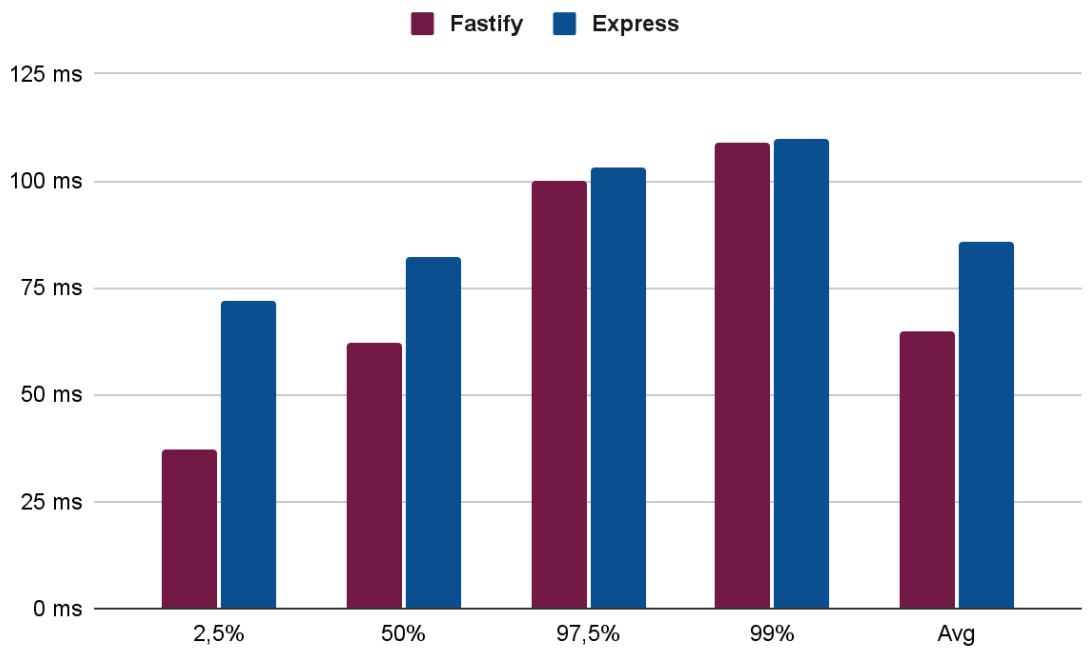
Avaliando a diferença nos valores das médias, o Fastify realizou em torno de 97,31 requisições por segundo a mais que o Express. O equilíbrio nos resultados também pode ser notado nos valores mínimos. Além disso, o desvio padrão indicado se apresentou como um pouco maior que nos testes realizados nas outras rotas,

esse desvio pode ser observado na diferença clara entre a quantidade de requisições feitas nos termos de percentis mais baixos e nos mais altos.

4.5 Rotas de remoção: latência e requisições por segundo

Nesta seção será realizada a análise dos resultados dos testes das rotas de remoção, que registraram os dados de latência e o número de requisições por segundo. As figuras a seguir mostram os resultados, e a análise dos mesmos oferece uma visão mais ampla e específica sobre o desempenho dos *frameworks* estudados.

Figura 18 - Representação gráfica dos testes de rotas de remoção referentes à latência

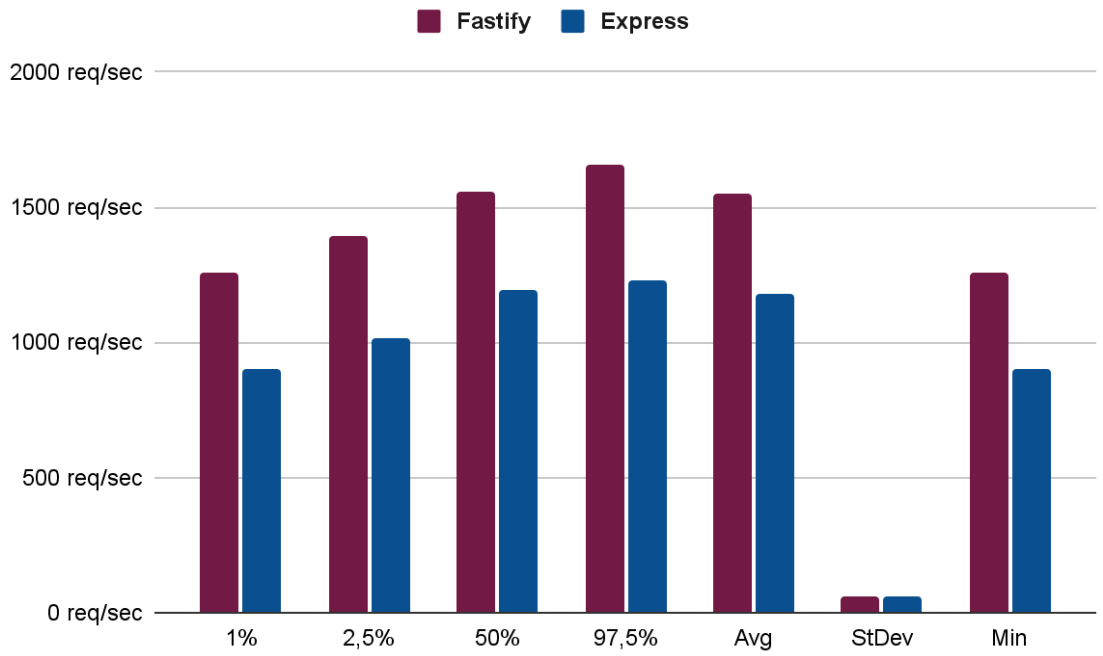


Fonte: Elaborado pelo autor, 2024

Na figura acima, visualiza-se os resultados dos testes de latência das rotas de remoção. A diferença entre os tempos de resposta é evidente no percentil de 2,5%, onde o Fastify foi quase duas vezes mais rápido que o Express. Essa diferença é reduzida no percentil de 50%, atingindo um equilíbrio claro nos percentis de 97,5% e 99%. Nos valores das médias de cada *framework* é possível observar por meio da

diferença de percentual que as requisições da aplicação com Fastify foram em média cerca de 24,3% mais rápidas que as da aplicação construída com Express.

Figura 19 - Representação gráfica dos testes de rotas de remoção referentes à requisições por segundo



Fonte: Elaborado pelo autor, 2024

A figura 15, mostra uma diferença constante em todos os termos de percentil referente à quantidade de requisições por segundo, com o Fastify em vantagem sobre o Express nesse aspecto. A maior diferença ocorreu no de 97,5%, onde o Fastify obteve 1657 e o Express 1230 requisições por segundo, totalizando 427 requisições por segundo a mais para o Express.

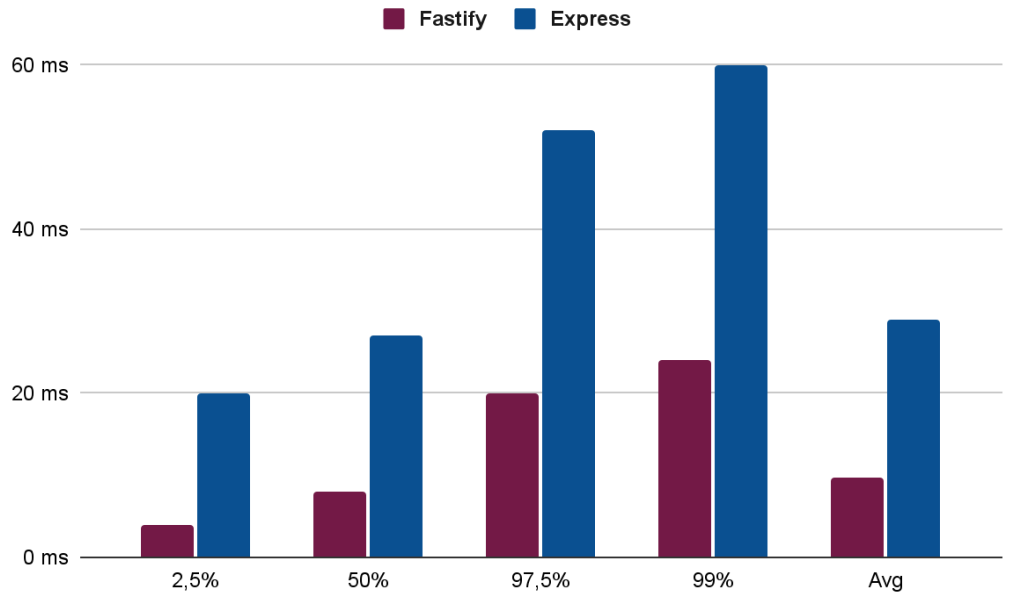
Essa diferença constante e sem uma discrepância tão grande entre os resultados de cada percentil, pode ser notada na observação do baixíssimo valor de desvio padrão. Já em termos de média, utilizando o cálculo do diferencial percentual, o Fastify realizou aproximadamente 31,92% de requisições por segundo a mais que o Express.

4.6 Rotas de status de API: latência, requisições por segundo, leitura de dados e quantidade de requisições

As rotas de status de API, assim como as rotas de listagem, utilizam o método GET. Porém, diferentemente das rotas de listagem que retornam todos os relatórios de *ping* presentes no banco de dados, as rotas de status retornam um dado de tamanho muito pequeno, sendo composto por um objeto contendo um valor booleano que representa o estado atual da API.

Este tamanho reduzido dos dados das respostas faz com que as requisições possuam um tempo de resposta muito pequeno, permitindo ao AutoCannon monitorar a quantidade de requisições totais realizadas de forma mais precisa, devido a isso, nesta seção, além dos parâmetros de latência, requisições por segundo e taxa de leitura de dados, também será abordado o aspecto da quantidade de requisições totais.

Figura 20 - Representação gráfica dos testes de rotas de status de API referentes à latência

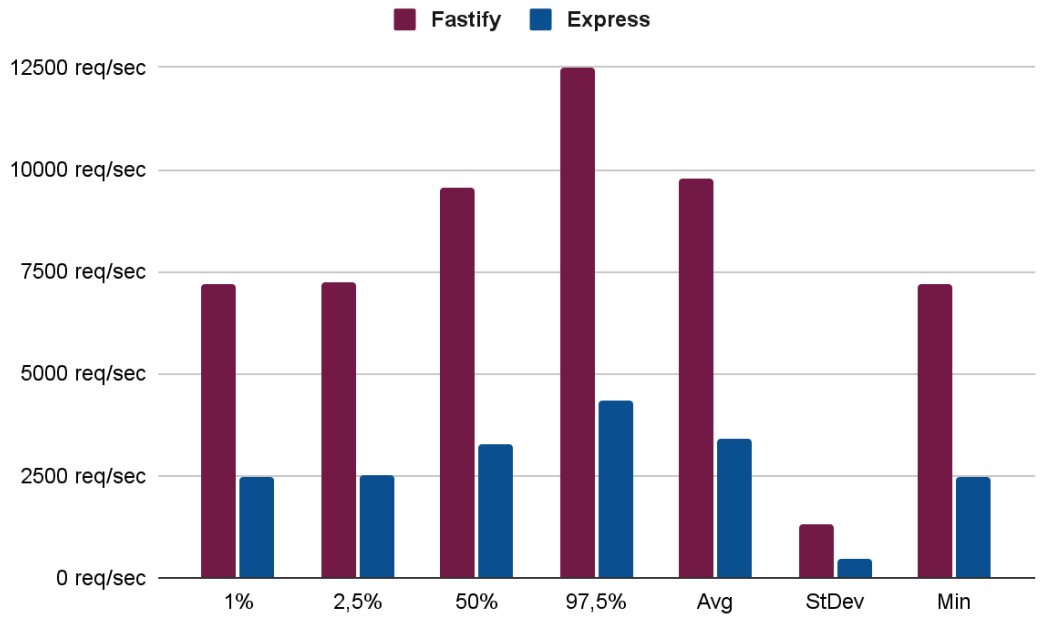


Fonte: Elaborado pelo autor, 2024

A figura 20, mostra os resultados dos testes de latência, onde a diferença dos tempos de resposta entre o Fastify e o Express é clara. No percentil de 2,5% o Fastify se mostrou cinco vezes mais rápido que o Express, tendo o primeiro

registrado 4 milissegundos e o segundo 20 milissegundos. No faixa de 50%, o Express apresentou 27 milissegundos e o Fastify 8 milissegundos, nas requisições lentas, representadas no percentil de 97,5%, o Express continuou em desvantagem, registrando 52 milissegundos enquanto o Fastify registrou 20 milissegundos. Por fim, a partir da observação dos valores de média o Fastify foi aproximadamente 66,52% mais rápido que o Express, mantendo a grande discrepância de valores.

Figura 21 - Representação gráfica dos testes de rotas de status de API referentes à requisições por segundo



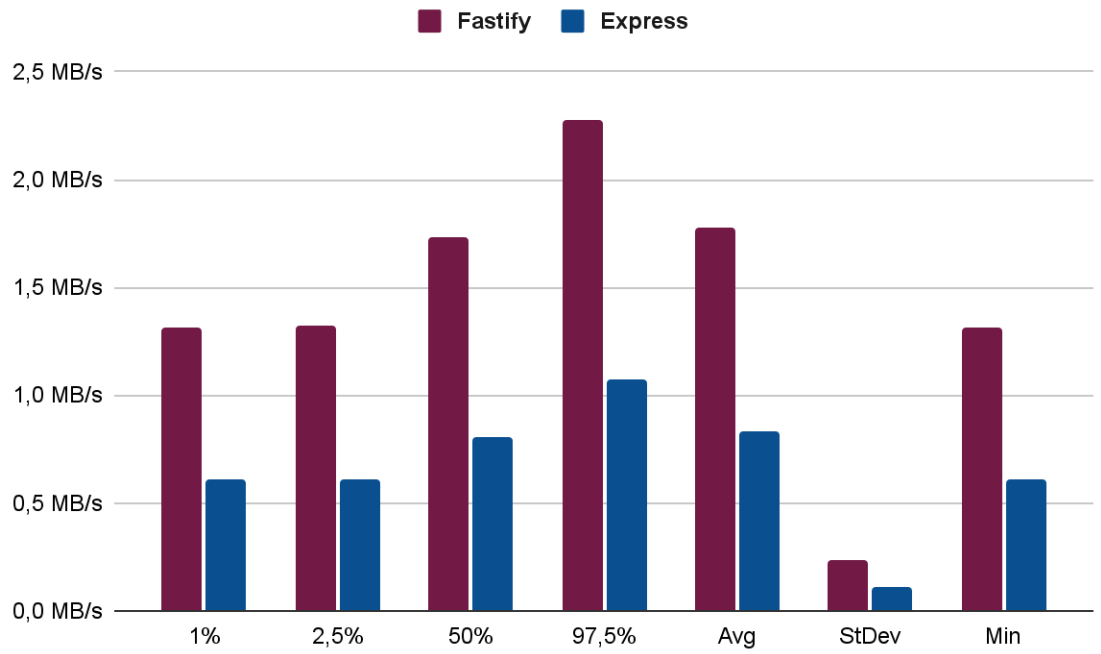
Fonte: Elaborado pelo autor, 2024

A imagem acima apresenta uma grande diferença de desempenho relativo ao parâmetro de quantidade de requisições por segundo, onde o Fastify claramente se sobressaiu sobre o Express nesse aspecto. Todos os percentis apresentam uma diferença considerável, sendo importante destacar os valores do percentil de 97,5%, tendo o Fastify registrado 12495 requisições por segundo, enquanto o Express registrou 4359. Nos outros percentis, assim como na média, o Fastify fez quase três vezes mais requisições por segundo que o Express.

Os valores apresentados na média foram: Fastify com 9782,34 requisições por segundo e Express com 3391,19. Essa discrepância também pode ser

observada nos valores da mínima. Em relação ao desvio padrão, o Fastify registrou um valor um pouco maior que o Express.

Figura 22 - Representação gráfica dos testes de rotas de status de API referentes à taxa de leitura de dados



Fonte: Elaborado pelo autor, 2024

A figura 22 contém os resultados relativos à taxa de leitura de dados. Observado o gráfico verifica-se que o Fastify registrou uma taxa de leitura maior que o Express em todas as medidas de percentis, assim como no valor mínimo. A maior diferença nos termos de percentil pode ser notada no percentil de 97,5%, onde o Fastify obteve 1,21 MB/s sobre o Express. Nos valores da média é possível perceber que o Fastify processou um pouco mais que o dobro de megabytes por segundo a mais que o Express.

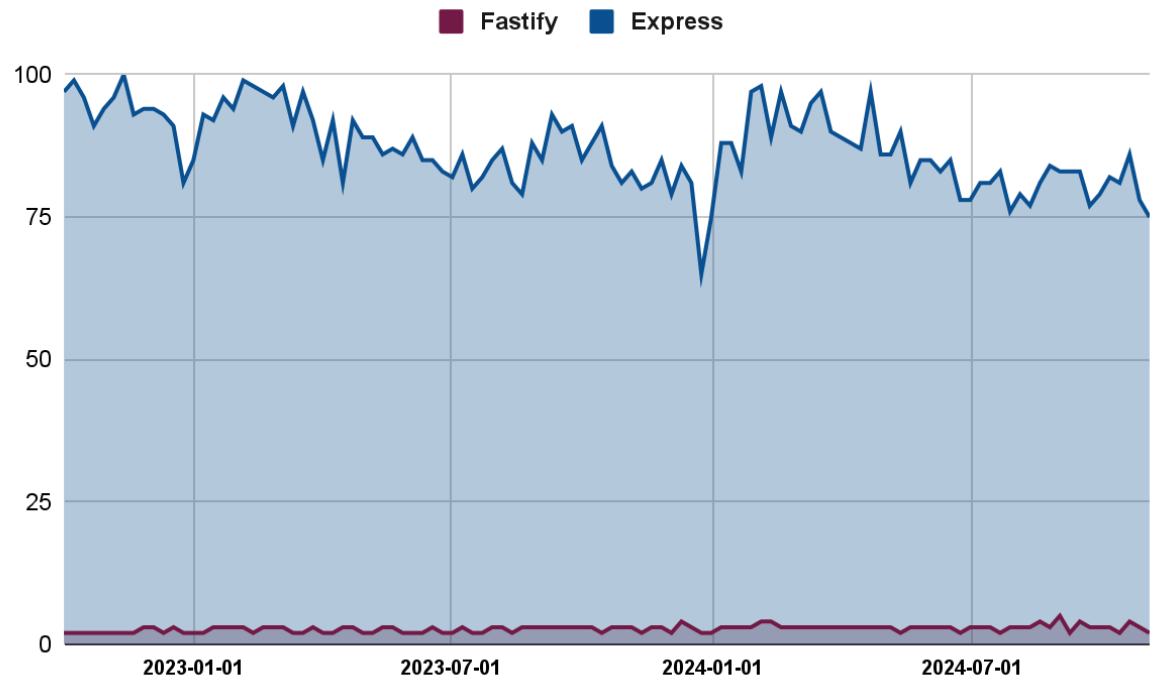
Partindo para o parâmetro do número total de requisições realizadas, que é abordado na análise das rotas de status de API, a aplicação com Fastify conseguiu realizar nos 60 segundos do teste um pouco mais que 587 mil requisições, enquanto o Express realizou pouco mais de 204 mil requisições.

4.7 Análise qualitativa: índices de popularidade

Para esta análise serão considerados alguns aspectos, sendo o primeiro deles, o interesse ao longo do tempo nos últimos dois anos (considerando a categoria de Ciência da Computação para uma melhor filtragem dos dados), obtido a partir da observação dos dados do Google Trends, que é uma ferramenta que permite acompanhar a evolução do número de buscas por um determinado tópico ou palavra-chave ao longo do tempo.

Além do parâmetro de interesse ao longo do tempo, serão observadas também a quantidade de estrelas registradas no repositório de cada *framework* no GitHub (ferramenta que permite o armazenamento e compartilhamento de código de fonte de um software) e a quantidade de *downloads* semanais de cada pacote no registrada no site do npm (Node Package Manager).

Figura 23 - Gráfico do interesse ao longo do tempo no período de dois anos obtido com dados do Google Trends



Fonte: Elaborado pelo autor, 2024

Observando os dados obtidos a partir do Google Trends, representados na figura 23, a diferença de popularidade dos dois *frameworks* é clara, onde em comparação com o Express, que registrou 87 pontos de busca na média, o Fastify registrou apenas 3 na média. Partindo para a análise dos dados de quantidade de estrelas no repositório do GitHub, a diferença de popularidade permanece evidente, com o Fastify contando com em torno de 32 mil estrelas, enquanto o Express registra aproximadamente 65 mil.

Por fim, observando o número de *download* semanais dos pacotes de cada *framework*, foi possível notar que há uma grande diferença nos números, onde Fastify registra um número aproximado de 1,7 milhões de *downloads*, enquanto o Express tem algo próximo de 32 milhões.

5 CONCLUSÃO

A análise dos dados, obtidos a partir dos testes e da observação dos índices de popularidade de cada *framework* analisado, mostra de forma clara que cada ferramenta apresenta benefícios na sua utilização, sendo possível concluir a partir disso, que o Fastify e o Express possuem situações apropriadas de uso.

De modo geral, é possível concluir que os objetivos que foram propostos para esta pesquisa foram alcançados. Ambas as aplicações foram implementadas de forma satisfatória e conseguiram ser utilizadas para a realização dos testes. Além disso, a execução dos testes foi bem sucedida e seus resultados possibilitaram a coleta de dados, que ao serem analisados, permitiram a obtenção de informações muito importantes para a avaliação dos desempenhos de cada ferramenta estudada.

5.1 Contribuições

Este trabalho contribui para o campo de desenvolvimento de APIs REST, fornecendo informações práticas para aqueles envolvidos e interessados na construção de aplicações web back-end, auxiliando na escolha do *framework* mais adequado de acordo com as suas necessidades. Mesmo considerando esse ponto, embora os testes realizados tenham obtido muitos dados úteis de comparação entre Express e Fastify, esta pesquisa não deve ser utilizada como única fonte de dados para a escolha final da tecnologia.

Através da análise dos dados obtidos a partir dos testes de desempenho, foi possível notar que o Fastify se mostrou mais eficiente nos aspectos de latência, processamento de requisições por segundo e taxa de leitura de dados (nos teste em que esse parâmetro foi observado) para a maioria dos cenários de teste. Já o Express, é uma opção amplamente utilizada, bem documentada e muito popular, possuindo um desempenho, que se mostrou ser de forma constante em grande parte dos testes realizados, inferior em comparação ao Fastify.

5.2 Limitações do estudo

Durante a produção deste estudo foram notados alguns fatores limitadores para o seu desenvolvimento. Um desses fatores está relacionado à forma como a qual o AutoCannon retorna os dados dos resultados, que ocorre de forma visual na linha de terminal, já que a ferramenta não possui uma funcionalidade que permita gerar gráficos ou qualquer tipo de representação visual dos dados. Devido a esse fator, a adição dos dados dos testes nos gráficos mostrados durante as análises dos resultados dos testes foi realizada de forma manual.

É muito importante destacar que por ter sido retornado de forma visual na linha de terminal, os dados extraídos foram, dentro da proposta da pesquisa, analisados e inseridos em suas respectivas representações gráfico-visuais de forma confiável e segura.

5.3 Trabalhos futuros

Um aspecto desta pesquisa que pode gerar novos trabalhos na área, é a comparação de performance entre *frameworks*, esse é um conceito que pode ser aplicado para *frameworks* que estão fora do ambiente Node.js e que consequentemente não utilizam JavaScript como sua linguagem de base, como por exemplo: Spring Boot (Java), Flask (Python), FastAPI (Python), Fiber (Go). Outra perspectiva que pode ser adotada no desenvolvimento de pesquisas futuras, está relacionada na comparação de desempenho de *frameworks* durante a execução de processamento de dados mais complexos, observando as diferentes maneiras utilizadas por cada plataforma para realizar tais tarefas, de modo geral expandindo a comparação para outros tipos de aplicação. Observando o aspecto da análise qualitativa desta pesquisa, seria interessante, como forma de reforçar esse ponto, a realização de um questionário (com a participação de indivíduos inseridos no contexto de desenvolvimento de uma API REST utilizando tecnologias do ambiente Node, como por exemplo, desenvolvedores web e estudantes da área de tecnologia da informação) com o objetivo de coletar informações sobre, por exemplo, suas experiências na utilização de ambas as tecnologias estudadas, seu nível de conhecimento sobre cada *framework* e seu interesse na utilização das ferramentas.

REFERÊNCIAS

BIERMAN, Gavin; ABADI, Martín; TORGERSEN, Mads. **Understanding TypeScript**. Proceedings of the 28th European Conference on ECOOP, 2014. Disponível em: https://link.springer.com/chapter/10.1007/978-3-662-44202-9_11. Acesso em: 21 out. 2024.

COLLINA, Matteo. **mcollina/autocannon: fast HTTP/1.1 benchmarking tool written in Node.js**. Disponível em: https://github.com/mcollina/autocannon. Acesso em: 24 maio 2024.

DAHL, Ryan. **10 Things I Regret About Node.js**. JSConf EU, 2018. Disponível em: https://www.youtube.com/watch?v=M3BM9TB-8yA. Acesso em: 27 nov. 2024.

DE ARAUJO, Luiz Eduardo Oliveira. **Análise comparativa do Tauri e Electron: um estudo de dois frameworks para aplicações de desktop multiplataforma**. Disponível em: https://dspace.bc.uepb.edu.br/jspui/handle/123456789/31022. Acesso em: 29 maio 2024.

DEMASHOV, Danil; GOSUDAREV, Ilya. **Efficiency Evaluation of Node.js Web-Server Frameworks**. CEUR Workshop Proceedings**,** 2023. Disponível em: https://ceur-ws.org/Vol-2590/short14.pdf. Acesso em: 18 maio 2024.

DI MEGLIO, Sergio; STARACE, Luigi Libero Lucio; DI MARTINO, Sergio. **Starting a New REST API project? A Performance Benchmark of Frameworks and Execution Environments**. CEUR Workshop Proceedings, 2019. Disponível em: https://ceur-ws.org/Vol-3543/paper19.pdf. Acesso em: 18 maio 2024.

EXPRESS. **Express: Framework web rápido, flexível e minimalista para Node.js**. Disponível em: https://expressjs.com/pt-br/. Acesso em: 08 junho 2024.

FACHIN, Odília. **Fundamentos da Metodologia**. 5. ed. São Paulo: Saraiva, 2006.

FASTIFY. **Fastify: Fast and low overhead web framework, for Node.js**. Disponível: https://fastify.dev/. Acesso em: 10 junho 2024.

GeeksforGeeks. **History of JavaScript.** Disponível em: https://www.geeksforgeeks.org/history-of-javascript/. Acesso em: 29 nov. 2024.

HÁMORI, Frenc. **History of Node.js on a Timeline.** Disponível em: https://blog.risingstack.com/history-of-node-js/. Acesso em: 28 nov. 2024.

HUANG, Xiaoping. **Research and Application of Node.js Core Technology**. IEEE Xplore, 2020. Disponível em: https://ieeexplore.ieee.org/abstract/document/9424850. Acesso em: 23 maio 2024.

HUMAND. **What is ExpressJS?.** Disponível em: https://humand.co.uk/what-is-expressjs/. Acesso em: 29 nov. 2024.

INVEDUS. **What is TypeScript? Definition, History, Features and Uses.** Disponível em: https://invedus.com/blog/what-is-typescript-definition-history-features-and-uses-of-typescript/. Acesso em: 30 nov. 2024.

JANSEN, Grave; SALADAS, Johanna. **Advantages of the event-driven architecture pattern**. IBM Developer, 2024. Disponível em: https://developer.ibm.com/articles/advantages-of-an-event-driven-architecture/. Acesso em: 25 maio 2024.

MARDAN, Azat. **Practical Node.js: Building Real-World Scalable Web Apps**. 1. ed. São Francisco: Apress, 2014. Acesso em: 29 maio 2024.

MOZILLA. **JavaScript.** Disponível em: https://developer.mozilla.org/pt-BR/docs. Acesso em: 29 maio 2024.

NODE. **Run JavaScript Everywhere**. Disponível em: https://nodejs.org/en. Acesso em: 23 maio 2024.

PRISMA. **Simplify working and interacting with databases.** Disponível em: https://www.prisma.io/. Acesso em: 22 set. 2024.

RELAN, Kunal. **Building REST APIs with Flask: Create Python Web Services with MySQL**. 1. ed. Nova Deli: Apress, 2019. Acesso em: 29 maio 2024.

ROCKETSEAT. **Configurando um servidor com Fastify**. Disponível em: https://blog.rocketseat.com.br/configurando-um-servidor-com-fastify/. Acesso em: 10 junho 2024.

STACK OVERFLOW. **Stack Overflow Developer Survey 2023**. Disponível em: https://survey.stackoverflow.co/2023/. Acesso em: 29 maio 2024.

STATE OF JAVASCRIPT. **State of JS 2022**. Disponível em: https://2022.stateofjs.com/. Acesso em: 10 junho 2024.

TYPESCRIPT. **TypeScript: JavaScript With Syntax For Types.** Disponível em: https://www.typescriptlang.org. Acesso em: 21 out. 2024.

WIKANDER, Daniel. **Exploring the quality attribute and performance implications of using GraphQL in a data-fetching API**. Digitala Vetenskapliga Arkivet, 2020. Disponível em: https://www.diva-portal.org/smash/get/diva2:1480750/FULLTEXT01.pdf. Acesso em: 24 maio 2024.

ZHOU, Wei et al. **REST API Design Patterns for SDN Northbound API**. 2014 28th International Conference on Advanced Information Networking and Applications Workshops, 2014. Disponível em: https://ieeexplore.ieee.org/document/6844664. Acesso em: 03 junho de 2024.