INSTITUTO FEDERAL DE PERNAMBUCO CAMPUS BELO JARDIM
BACHARELADO EM ENGENHARIA DE SOFTWARE

MARIA GABRIELLY DE ALMEIDA ARAÚJO

ANÁLISE COMPARATIVA DE DESEMPENHO DE APIS RESTFUL EM NODE.JS, .NET E GO COM K6

Belo Jardim, Pernambuco
2024

MARIA GABRIELLY DE ALMEIDA ARAÚJO

ANÁLISE COMPARATIVA DE DESEMPENHO DE APIS RESTFUL EM NODE.JS, .NET E GO COM K6

Trabalho de conclusão de Curso (TCC) apresentado como requisito parcial para obtenção do grau de Bacharel em Engenharia de Software.

Banca de Qualificação:

Jobson Tenório do Nascimento - IFPE - Campus Belo Jardim
Elton Bezerra Torres - IFPE - Campus Belo Jardim
José Fernando da Silva - IFPE - Campus Garanhuns

Belo Jardim, Pernambuco
2024

Dados Internacionais de Catalogação - CIP

A663a Araújo, Maria Gabrielly de Almeida
Análise comparativa de desempenho de APIs RESTful em Node.js, .NET e Go com K6 / Maria Gabrielly de Almeida Araújo. – Belo Jardim-PE, 2024.
86f.: il.

Trabalho de Conclusão de Curso (Bacharelado em Engenharia de Software) – Instituto Federal de Educação, Ciência e Tecnologia de Pernambuco, Campus Belo Jardim- PE, 2024.

Orientador: Prof.º Jobson Tenório do Nascimento.

Inclui referências.

1. Desenvolvimento de software. 2. Aplicações web - desempenho. 3. Tecnologia da Informação. 4. Sistemas de informação. I. Título. II. Nascimento, Jobson Tenório do. III. Instituto Federal de Educação, Ciência e Tecnologia de Pernambuco.

CDD 005

Ficha catalográfica elaborada pelo Sistema Integrado de Bibliotecas - SIBI/IFPE.
Bibliotecária: Louise Machado Freire Dias CRB – 4 /2267.

MARIA GABRIELLY DE ALMEIDA ARAÚJO

ANÁLISE COMPARATIVA DE DESEMPENHO DE APIS RESTFUL EM NODE.JS, .NET E GO COM K6

Trabalho aprovado. Belo Jardim, 06/12/2024.

Jobson Tenório do Nascimento

Professor Orientador

Elton Bezerra Torres

Convidado 1

José Fernando da Silva

Convidado 2

Belo Jardim, Pernambuco
2024

AGRADECIMENTOS

Agradeço a Deus, sem Ele nada é possível. Toda honra e glória pertencem a Ele.

Aos meus pais, Maria Aparecida e José Francisco, e à minha irmã, Marielly, por todo apoio e incentivo para não desistir em meio aos desafios durante essa jornada.

Aos professores José Fernando e Jobson Tenório pela confiança, todo conhecimento compartilhado e a liberdade para realizar esta pesquisa. Também expresso minha gratidão ao professor Elton, membro da banca avaliadora, e a todos os outros professores que estiveram em minha jornada acadêmica, contribuindo para o meu desenvolvimento.

Por fim, muitas pessoas me ajudaram, não apenas neste trabalho, mas ao longo de toda caminhada, e para não deixar de citar ninguém, quero que saibam que sempre em meu coração irei ser grata, pois ninguém consegue conquistar nada sozinho. Agradeço por cada conversa, apoio, risadas e por todos os momentos que me fizeram crescer, tanto profissionalmente quanto pessoalmente.

“Tudo tem o seu tempo determinado, e há tempo para todo o propósito debaixo do céu:
Tempo de nascer e tempo de morrer, tempo de plantar e tempo de arrancar o que se plantou.”

Eclesiastes 3:1-2

RESUMO

Este trabalho teve como objetivo realizar uma análise comparativa de desempenho de APIs RESTful desenvolvidas com Node.js, .NET e Go, utilizando a ferramenta k6 para medir o tempo de resposta em diferentes cenários de carga. A metodologia adotada foi de caráter experimental, com a implementação das APIs em cada uma das tecnologias mencionadas e a execução de testes controlados em um ambiente estável. Os resultados obtidos mostraram que o ASP.NET Core apresentou o melhor desempenho sob altas cargas de requisições, enquanto o Node.js foi eficiente em cenários de menor carga. A tecnologia Go, por sua vez, demonstrou um equilíbrio entre desempenho e simplicidade, sendo adequada para aplicações que exigem escalabilidade moderada. Conclui-se que a escolha da tecnologia para o desenvolvimento de APIs RESTful deve considerar o contexto específico de uso e a demanda de requisições simultâneas, uma vez que cada plataforma possui vantagens distintas em termos de desempenho e escalabilidade.

Palavras-chave: APIs RESTful; Desempenho; Node.js; .NET; Go.

ABSTRACT

This work aimed to perform a comparative analysis of the performance of RESTful APIs developed with Node.js, .NET, and Go, using the k6 tool to measure response times in different load scenarios. The adopted methodology was experimental, involving the implementation of the APIs in each of the mentioned technologies and the execution of controlled tests in a stable environment. The results showed that ASP.NET Core exhibited the best performance under high request loads, while Node.js was efficient in lower load scenarios. Go, on the other hand, demonstrated a balance between performance and simplicity, making it suitable for applications that require moderate scalability. It is concluded that the choice of technology for developing RESTful APIs should consider the specific usage context and the demand for simultaneous requests, as each platform has distinct advantages in terms of performance and scalability.

Keywords: RESTful APIs; Performance; Node.js; .NET; Go.

LISTA DE FIGURAS

Figura 1 - Número de usuários da Internet no mundo no período de 2005 a 2023. 15
Figura 2 - Modelo arquitetural REST. 24
Figura 3 - Estrutura de uma URL. 26
Figura 4 - Representação de um JSON. 27
Figura 5 - Diagrama de Entidade-Relacionamento. 42
Figura 6 - Método POST no Controller de Livros. 44
Figura 7 - Código do método create no BookService. 44
Figura 8 - Código do método POST no Repository de Livros. 45
Figura 9 - Método POST no BooksController. 47
Figura 10 - Método UpdateBook no BookService. 48
Figura 11 - Método GetBookById no BookRepository. 49
Figura 12 - Método GetBooks no BookHandler. 52
Figura 13 - Método GetBooks no BookService. 52
Figura 14 - Método FindAll no BookRepository. 53
Figura 15 - Mind Map dos cenários de teste por tipo de requisição e seus respectivos casos por carga de usuários. 55
Figura 16 - Script do teste do caso 1, com 10 usuários no cenário 1, requisição POST. 58
Figura 17 - Script do teste do caso 1, com 10 usuários no cenário 2, requisição GET. 59
Figura 18 - Script do teste do caso 1, com 10 usuários no cenário 3, requisição PUT. 61
Figura 19 - Script do teste do caso 1, com 10 usuários no cenário 4, requisição DELETE. 62
Figura 20 - Exemplo do comando de execução do teste com k6. 64
Figura 21 - Resultado exibido no terminal do teste no cenário 2 (GET) para o caso 1 (10 usuários). 64
Figura 22 - Tempo Médio, Mínimo e Máximo de Resposta das APIs com 10 Usuários Simultâneos no Cenário 1. 66
Figura 23 - Tempo Médio, Mínimo e Máximo de Resposta das APIs com 100 Usuários Simultâneos no Cenário 1. 67
Figura 24 - Tempo Médio, Mínimo e Máximo de Resposta das APIs com 500 Usuários Simultâneos no Cenário 1. 68
Figura 25 - Tempo Médio, Mínimo e Máximo de Resposta das APIs com 1000 Usuários Simultâneos no Cenário 1. 69
Figura 26 - Tempo Médio, Mínimo e Máximo de Resposta das APIs com 10 Usuários Simultâneos no Cenário 2. 70
Figura 27 - Tempo Médio, Mínimo e Máximo de Resposta das APIs com 100 Usuários Simultâneos no Cenário 2. 71
Figura 28 - Tempo Médio, Mínimo e Máximo de Resposta das APIs com 500 Usuários Simultâneos no Cenário 2. 72
Figura 29 - Tempo Médio, Mínimo e Máximo de Resposta das APIs com 1000 Usuários Simultâneos no Cenário 2. 73
Figura 30 - Tempo Médio, Mínimo e Máximo de Resposta das APIs com 10 Usuários Simultâneos no Cenário 3 74

Figura 31 - Tempo Médio, Mínimo e Máximo de Resposta das APIs com 100 Usuários Simultâneos no Cenário 3. 75
Figura 32 - Tempo Médio, Mínimo e Máximo de Resposta das APIs com 500 Usuários Simultâneos no Cenário 3. 76
Figura 33 - Tempo Médio, Mínimo e Máximo de Resposta das APIs com 1000 Usuários Simultâneos no Cenário 3. 77
Figura 34 - Tempo Médio, Mínimo e Máximo de Resposta das APIs com 10 Usuários Simultâneos no Cenário 4. 78
Figura 35 - Tempo Médio, Mínimo e Máximo de Resposta das APIs com 100 Usuários Simultâneos no Cenário 4. 79
Figura 36 - Tempo Médio, Mínimo e Máximo de Resposta das APIs com 500 Usuários Simultâneos no Cenário 4. 80
Figura 37 - Tempo Médio, Mínimo e Máximo de Resposta das APIs com 1000 Usuários Simultâneos no Cenário 4. 81

LISTA DE QUADROS

Quadro 1 - Quadro metodológico da pesquisa. 35

LISTA DE TABELAS

Tabela 1 - Principais métodos HTTP 25
Tabela 2 - Categorias de códigos de status HTTP 25

LISTA DE ABREVIATURAS E SIGLAS

API	*Application Programming Interface*
ASP.NET CORE	Framework web da plataforma .NET
CPU	*Central Processing Unit*
DELETE	Método HTTP para exclusão de dados
ECHO	Framework para Go
GET	Método HTTP para leitura de dados
GO	Linguagem de programação
GORM	*Go Object-Relational Mapping*
HTTP	*Hypertext Transfer Protocol*
I/O	*Input/Output*
JSON	*JavaScript Object Notation*
k6	Ferramenta de teste de carga desenvolvida em Go
NESTJS	Framework para desenvolvimento no ambiente Node.js
NPM	*Node Package Manager*
NODE.JS	Ambiente de execução JavaScript do lado do servidor
ORM	*Object-Relational Mapping*
PATCH	Método HTTP para atualização parcial
POST	Método HTTP para envio de dados
PRISMA	ORM para Node.js
PUT	Método HTTP para atualização de dados
RAM	*Random Access Memory*
REST	*Representational State Transfer*
SQL	*Structured Query Language*
SSD	*Solid State Drive*
TLS	*Transport Layer Security*
URL	*Uniform Resource Locator*
V8	Motor JavaScript de alta performance
XML	*Extensible Markup Language*

SUMÁRIO

1. INTRODUÇÃO **15**
1.1 CONTEXTUALIZAÇÃO 15
1.2 PROBLEMÁTICA DA PESQUISA 16
1.3 MOTIVAÇÃO E JUSTIFICATIVA 16
1.4 OBJETIVOS 17
1.4.1 Objetivo Geral **17**
1.4.2 Objetivos Específicos **17**
1.5 ESTRUTURA DA PESQUISA 18
2. FUNDAMENTAÇÃO TEÓRICA **19**
2.1 APPLICATION PROGRAMMING INTERFACE (API) 19
2.1.1 Classificação por Tipo de Interface **19**
2.1.1.1 APIs de Hardware 19
2.1.1.2 APIs de Bibliotecas 20
2.1.1.3 APIs de Sistemas Operacionais 20
2.1.1.4 Web APIs 20
2.2 API RESTful 21
2.2.1 Arquitetura REST **22**
2.2.2 Protocolo HTTP **24**
2.2.3 Métodos HTTP **24**
2.2.4 Códigos de Status HTTP **25**
2.2.5 Endpoints **26**
2.2.6 Representação de Dados com JSON **27**
2.3 NODE.JS 28
2.3.1 O Motor V8 **28**
2.3.2 Framework NestJS **29**
2.4 .NET 30
2.4.1 Framework ASP.NET Core **30**
2.5 GO 31
2.5.1 Framework Echo **32**
2.6 FERRAMENTA DE TESTE DE DESEMPENHO 32
2.6.1 k6 **33**
2.7 TRABALHOS RELACIONADOS 33
3. METODOLOGIA **35**
3.1 CARACTERIZAÇÃO DA PESQUISA 35
3.1.1 Quanto à Finalidade **35**
3.1.2 Quanto à Natureza **36**
3.1.3 Quanto à Abordagem **36**
3.1.4 Quanto ao Objetivo **36**
3.1.5 Quanto a Estratégia **37**

3.1.6 Quanto ao Método Científico **37**
3.1.7 Quanto aos Procedimentos Técnicos **38**
3.1.8 Quanto aos Procedimentos para coleta de dados **38**
3.2 MÉTODO DE PESQUISA 38
3.2.1 Método Experimental **39**
3.3 EXPERIMENTO DA PESQUISA 39
3.3.1 Ambiente Controlado **39**
3.3.2 Métrica analisada **40**
3.4 MÉTODOS E TÉCNICAS DE COLETA E ANÁLISE DE DADOS 40
3.4.1 Observação sistemática **40**
4. IMPLEMENTAÇÃO **41**
4.1 DESENVOLVIMENTO DAS APIs 41
4.1.1 Descrição e Funcionalidades das APIs **41**
4.1.2 Banco de Dados **42**
4.1.2.1 Diagrama de Entidade-Relacionamento 42
4.1.3 API em Node.js com NestJS **43**
4.1.3.1 Arquitetura da API 43
4.1.3.1.1 Controllers 43
4.1.3.1.2 Services 44
4.1.3.1.3 Repositories 45
4.1.3.2 Estrutura do Projeto 45
4.1.4 API em ASP.NET Core **47**
4.1.4.1 Arquitetura da API 47
4.1.4.1.1 Controllers 47
4.1.4.1.2 Services 48
4.1.4.1.3 Repositories 49
4.1.4.2 Estrutura do Projeto 50
4.1.5 API em Go com Echo **51**
4.1.5.1 Arquitetura da API 51
4.1.5.1.1 Handlers 51
4.1.5.1.2 Services 52
4.1.5.1.3 Repositories 52
4.1.5.2 Estrutura do Projeto 53
4.1.6 Repositório dos Códigos **54**
4.2 CENÁRIOS DE TESTE 55
4.2.1 Cenário 1: Requisição POST **56**
4.2.2 Cenário 2: Requisição GET **56**
4.2.3 Cenário 3: Requisição PUT **56**
4.2.4 Cenário 4: Requisição DELETE **57**
4.3 DESENVOLVIMENTO DOS SCRIPTS DE TESTE 57
4.3.1 Script do Cenário 1: Requisição POST **57**

4.3.2 Script do Cenário 2: Requisição GET **59**
4.3.3 Script do Cenário 3: Requisição PUT **60**
4.3.4 Script do Cenário 4: Requisição DELETE **62**
4.4 EXECUÇÃO DOS TESTES 63
5. RESULTADOS **66**
5.1 RESULTADOS DO CENÁRIO 1: MÉTODO POST 66
5.1.1 Resultados do Caso 1: 10 Usuários Simultâneos **66**
5.1.2 Resultados do Caso 2: 100 Usuários Simultâneos **67**
5.1.3 Resultados do Caso 3: 500 Usuários Simultâneos **68**
5.1.4 Resultados do Caso 4: 1000 Usuários Simultâneos **69**
5.2 RESULTADOS DO CENÁRIO 2: MÉTODO GET 70
5.2.1 Resultados do Caso 1: 10 Usuários Simultâneos **70**
5.2.2 Resultados do Caso 2: 100 Usuários Simultâneos **71**
5.2.3 Resultados do Caso 3: 500 Usuários Simultâneos **72**
5.2.4 Resultados do Caso 4: 1000 Usuários Simultâneos **73**
5.3 RESULTADOS DO CENÁRIO 3: MÉTODO PUT 74
5.3.1 Resultados do Caso 1: 10 Usuários Simultâneos **74**
5.3.2 Resultados do Caso 2: 100 Usuários Simultâneos **75**
5.3.3 Resultados do Caso 3: 500 Usuários Simultâneos **76**
5.3.4 Resultados do Caso 4: 1000 Usuários Simultâneos **77**
5.4 RESULTADOS DO CENÁRIO 4: MÉTODO DELETE 78
5.4.1 Resultados do Caso 1: 10 Usuários Simultâneos **78**
5.4.2 Resultados do Caso 2: 100 Usuários Simultâneos **79**
5.4.3 Resultados do Caso 3: 500 Usuários Simultâneos **80**
5.4.4 Resultados do Caso 4: 1000 Usuários Simultâneos **81**
5.5 ANÁLISE COMPARATIVA 82
6. CONCLUSÕES **84**
6.1 CONSIDERAÇÕES FINAIS 84
6.2 LIMITAÇÕES DA PESQUISA 84
6.3 SUGESTÕES PARA TRABALHOS FUTUROS 85
REFERÊNCIAS **86**
APÊNDICES **91**
APÊNDICE A - Scripts de testes do cenário POST **91**
APÊNDICE B - Scripts de testes do cenário GET **95**
APÊNDICE C - Scripts de testes do cenário PUT **97**
APÊNDICE D - Scripts de testes do cenário DELETE **101**

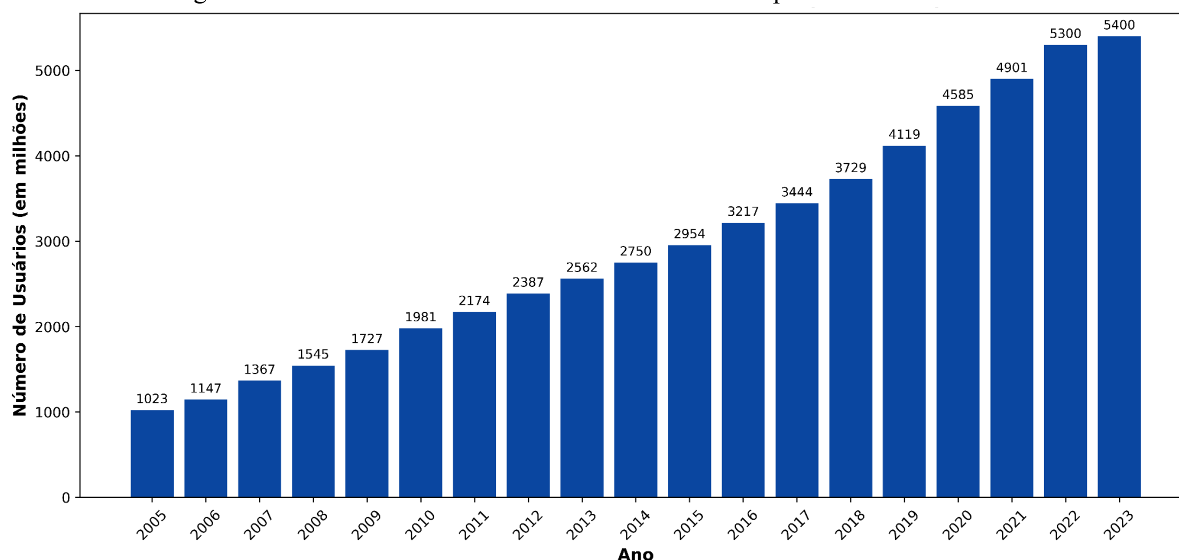
1. INTRODUÇÃO

Neste primeiro capítulo, será apresentada a contextualização do tema de pesquisa e a problemática investigada. Além disso, serão expostas as motivações e justificativas que levaram à escolha do tema, assim como os objetivos gerais e específicos da pesquisa, seguidos pela estrutura do trabalho.

1.1 CONTEXTUALIZAÇÃO

Nas últimas décadas, a internet passou por um crescimento exponencial tanto em termos de usuários conectados quanto na quantidade de dados trafegados. Como ilustrado na Figura 1, o número de pessoas usando a internet aumentou significativamente, passando de aproximadamente 1 bilhão de usuários em 2005 para mais de 5 bilhões em 2023, esse crescimento tem sido impulsionado por diversas inovações tecnológicas e pela crescente adoção de dispositivos móveis, plataformas web e serviços conectados (BITTENCOURT, 2021; PETROSYAN, 2024).

Figura 1 - Número de usuários da Internet no mundo no período de 2005 a 2023.



Fonte: Adaptada de PETROSYAN (2024).

Nesse contexto, a demanda por sistemas e aplicações capazes de suportar grandes volumes de tráfego e fornecer respostas rápidas e escaláveis também cresceu. As APIs RESTful, fundamentais para a comunicação entre sistemas distribuídos, desempenham um papel crucial nesse cenário, facilitando a integração de diferentes serviços e garantindo uma experiência de usuário fluida (SHKODRA; JAJAGA; SHALA, 2021).

A escolha da tecnologia para o desenvolvimento de APIs RESTful é um fator determinante para o sucesso de aplicações web, pois impacta diretamente o desempenho, a escalabilidade e a manutenção dos sistemas (SHKODRA; JAJAGA; SHALA, 2021). Tecnologias diferentes, como Node.js, .NET e Go, oferecem características únicas que podem atender a necessidades específicas de negócios e cenários de alta demanda.

O desempenho dessas APIs não apenas afeta a eficiência operacional das aplicações, mas também influencia diretamente a satisfação do cliente. Assim, a escolha da tecnologia ideal pode resultar em economias significativas de custo e na melhoria da experiência do usuário, tornando-se um fator crítico para os profissionais de TI ao tomarem decisões estratégicas (DALBARD; ISACSON, 2021).

1.2 PROBLEMÁTICA DA PESQUISA

No cenário atual, as aplicações web dependem fortemente de APIs RESTful para garantir comunicação eficiente entre sistemas. No entanto, cada tecnologia utilizada para desenvolver essas APIs oferece características distintas, que podem afetar significativamente o desempenho das aplicações, especialmente em ambientes com alta demanda de requisições (SHKODRA; JAJAGA; SHALA, 2021). Ainda que existam estudos comparativos entre linguagens de programação e frameworks para desenvolvimento web, faltam análises focadas na comparação no desempenho de APIs RESTful construídas com Node.js, .NET e Go sob condições de carga.

Profissionais de TI necessitam de *insights* baseados em dados para tomar decisões informadas sobre a tecnologia a ser adotada, uma vez que essa escolha pode influenciar a escalabilidade, a manutenção e, consequentemente, o sucesso das aplicações em um ambiente digital em rápida evolução (DALBARD; ISACSON, 2021). Assim, este estudo busca responder à seguinte questão: **"Qual tecnologia, entre Node.js, .NET e Go, oferece o melhor desempenho em termos de tempo de resposta de APIs RESTful?"** Esta pergunta é fundamental, pois o tempo de resposta das APIs não apenas afeta a eficiência operacional, mas também a experiência do usuário e a satisfação do cliente.

1.3 MOTIVAÇÃO E JUSTIFICATIVA

A motivação para esta pesquisa surge da crescente importância das APIs RESTful no desenvolvimento de aplicações web, onde a eficiência e o desempenho são cruciais para o sucesso das empresas. Com a digitalização em expansão, as APIs são fundamentais para a

integração entre sistemas, em um mercado cada vez mais competitivo, em que a rapidez no processamento de requisições e a escalabilidade das aplicações impactam diretamente a experiência do usuário (BITTENCOURT, 2021; CARMO, 2023). Assim, entender como diferentes tecnologias afetam o desempenho das APIs RESTful é essencial para que as organizações façam escolhas estratégicas que maximizem sua eficiência.

Além disso, a justificativa para a escolha dessas três tecnologias é relevante devido às suas características distintas. Node.js é amplamente utilizado por sua natureza assíncrona e alto desempenho em aplicações em tempo real, enquanto o .NET, com sua robustez e suporte a grandes projetos corporativos, é uma escolha popular entre desenvolvedores (SHKODRA; JAJAGA; SHALA, 2021). Por outro lado, o Go se destaca pela simplicidade e eficiência em ambientes de alta concorrência (GO, 2019). Compreender como cada uma dessas tecnologias se comporta em termos de desempenho pode oferecer *insights* valiosos para a adoção adequada em projetos futuros.

1.4 OBJETIVOS

A pesquisa foi estruturada com dois níveis de objetivos: o geral e os específicos. O objetivo geral define a meta principal, enquanto os objetivos específicos detalham as submetas necessárias para alcançá-la.

1.4.1 Objetivo Geral

A presente pesquisa tem como objetivo geral avaliar e comparar o desempenho de APIs RESTful desenvolvidas em Node.js, .NET e Go, utilizando o k6 para medir o tempo de resposta das requisições.

1.4.2 Objetivos Específicos

Para alcançar o objetivo geral proposto nesta dissertação, buscam-se atingir os seguintes objetivos específicos:

1. Implementar APIs RESTful em Node.js, .NET e Go, utilizando frameworks adequados para cada tecnologia (NestJS, ASP.NET Core e Echo, respectivamente).
2. Configurar e realizar experimentos controlados utilizando o k6 para medir o desempenho das APIs RESTful desenvolvidas sob diferentes tecnologias.

3. Coletar e analisar dados de desempenho, especificamente o tempo de resposta das requisições para cada uma das três implementações.
4. Realizar uma análise estatística dos dados coletados para comparar o desempenho entre as três tecnologias e identificar qual oferece o melhor desempenho em termos de tempo de resposta e eficiência sob diferentes cargas de trabalho.

1.5 ESTRUTURA DA PESQUISA

Este trabalho está estruturado em seis capítulos, conforme descrito a seguir:

- Capítulo 1 - Introdução: apresenta a contextualização, problemática, motivação e justificativa da pesquisa, além dos objetivos e a estrutura do trabalho.
- Capítulo 2 - Fundamentação Teórica: descreve os conceitos fundamentais sobre APIs RESTful, as tecnologias Node.js, .NET e Go, e a ferramenta de teste de desempenho, o k6.
- Capítulo 3 - Metodologia: detalha os métodos e técnicas utilizados para conduzir os experimentos, incluindo a coleta e análise dos dados.
- Capítulo 4 - Implementação: explica o processo de desenvolvimento das APIs RESTful nas três tecnologias e a configuração dos testes de desempenho.
- Capítulo 5 - Resultados: apresenta os dados coletados nos experimentos e a análise comparativa de desempenho entre as três implementações.
- Capítulo 6 - Conclusões: resume as principais conclusões da pesquisa, destacando as contribuições e propondo sugestões para trabalhos futuros.

2. FUNDAMENTAÇÃO TEÓRICA

Neste capítulo, será abordada toda a base teórica e tecnológica necessária para o entendimento deste estudo. O foco estará em três áreas principais: as Interfaces de Programação de Aplicações (APIs), com ênfase nas APIs RESTful; os ambientes de desenvolvimento Node.js, utilizando o framework NestJS, e .NET, com o ASP.NET Core, para construir duas das APIs; e a linguagem de programação Go, com o framework Echo, utilizado para criar a terceira API. Além disso, serão discutidas ferramentas de teste de desempenho, especificamente o k6.

2.1 APPLICATION PROGRAMMING INTERFACE (API)

Uma API (Application Programming Interface), ou Interface de Programação de Aplicações, é uma interface fundamental na programação que facilita a comunicação entre sistemas e aplicações diferentes por meio de um conjunto de protocolos e ferramentas de desenvolvimento. Em termos simples, uma API atua como uma ponte entre diferentes softwares, permitindo que eles interajam e compartilhem dados sem a necessidade de entender a complexidade interna uns dos outros (GOODWIN, 2024).

Essencial para a integração de sistemas, a API define claramente como as solicitações e respostas devem ser estruturadas, garantindo que a troca de informações entre sistemas seja feita de forma eficiente e compreensível (CARMO, 2023). Com o surgimento das APIs, grandes empresas começaram a conectar diversas aplicações e serviços de forma mais fluida, possibilitando uma maior interoperabilidade e uma troca de dados mais simplificada, independentemente das tecnologias e plataformas utilizadas (SOUZA et al., 2020).

2.1.1 Classificação por Tipo de Interface

As APIs podem ser classificadas de acordo com o tipo de interface que oferecem, pois ajuda a entender as diferentes formas de interação que proporcionam e como elas se integram a sistemas e aplicações (RAZA, 2023). Abaixo estão apresentados os principais tipos de interface de APIs:

2.1.1.1 APIs de Hardware

APIs de hardware proporcionam uma interface que permite ao software interagir com componentes físicos do sistema, como CPUs e placas de vídeo, de forma padronizada. Elas

abstraem os detalhes técnicos do hardware, permitindo aos desenvolvedores controlar e utilizar suas funcionalidades sem precisar se aprofundar nas complexidades internas. Essas APIs são fundamentais para garantir que as aplicações possam operar de forma eficiente e integrada com diversos dispositivos; por exemplo a DirectX, que oferece suporte para múltiplos tipos de hardware em ambientes Windows (LOUZADA; CARVALHO; LARANJA, 2024).

2.1.1.2 APIs de Bibliotecas

APIs de biblioteca são conjuntos de funções e classes prontas que facilitam tarefas específicas em projetos de software, permitindo a reutilização de código e simplificando o desenvolvimento. Como exemplo, pode-se citar o Matplotlib, que ajuda a criar e visualizar gráficos em Python, e *requests*, que simplifica requisições HTTP. Essas APIs permitem que os desenvolvedores integrem funcionalidades complexas sem precisar programar cada detalhe do zero, agilizando o processo de desenvolvimento e evitando a reinvenção de soluções comuns (LOUZADA; CARVALHO; LARANJA, 2024).

2.1.1.3 APIs de Sistemas Operacionais

As APIs de sistemas operacionais permitem que aplicativos interajam com o sistema subjacente, facilitando tarefas como manipulação de arquivos e configuração do sistema. Cada sistema operacional, como Windows ou Linux, possui seu próprio conjunto de APIs, que inclui interfaces para acessar recursos e serviços do sistema, fazendo com que os aplicativos utilizem eficientemente os recursos do sistema sem interagir diretamente com o hardware ou o núcleo do sistema operacional (LOUZADA; CARVALHO; LARANJA, 2024).

2.1.1.4 Web APIs

As APIs web atuam como uma ponte entre clientes e servidores, utilizando protocolos como HTTP para facilitar a comunicação pela internet, isso permite que aplicações distintas interajam e compartilhem dados de forma padronizada, independentemente das tecnologias envolvidas. O funcionamento típico envolve o cliente enviando uma requisição HTTP a um endpoint no servidor, que processa a solicitação e retorna uma resposta, normalmente em formato JSON ou XML (LOUZADA; CARVALHO; LARANJA, 2024). As web APIs podem ser classificadas com base em suas arquiteturas ou protocolos. Abaixo estão listadas algumas das principais e mais consolidadas no mercado:

- REST (Representational State Transfer): Desenvolvido nos anos 2000, é uma arquitetura que usa métodos HTTP para manipular recursos identificados por URLs. O cliente envia dados ao servidor, que processa e retorna respostas em formatos como JSON ou XML, acompanhadas de códigos HTTP (CHINA, 2024).
- GraphQL: Arquitetura que permite consultas flexíveis e específicas, permitindo que o cliente solicite exatamente os dados necessários. Ao contrário do REST, que utiliza múltiplos endpoints, o GraphQL permite obter todos os dados em uma única consulta, utilizando *resolvers* para transformar e reunir informações de diversas fontes (CHINA, 2024).
- RPC (Remote Procedure Call): Protocolo que permite a execução de funções remotamente como se fossem locais, usando HTTP para a comunicação. O cliente envia dados serializados ao servidor, que processa e retorna o resultado, com stubs gerenciando a conversão dos dados (MIAZAKI, 2021).

2.2 API RESTful

As APIs RESTful são uma forma de interface que facilita a comunicação entre sistemas de computador através da internet, utilizando padrões amplamente aceitos como o protocolo HTTP e HTTPS. Essas APIs seguem um conjunto de princípios arquitetônicos definidos pela arquitetura REST, que se caracteriza por sua independência e simplicidade na troca de informações (CARMO, 2023). O principal objetivo das APIs RESTful é proporcionar um método eficiente e seguro para a interação entre sistemas distribuídos, permitindo a comunicação de dados de forma desacoplada e confiável (GOLMOHAMMADI; ZHANG; ARCURI, 2023).

Em vez de manter o estado entre requisições, as APIs RESTful tratam cada solicitação de forma independente, simplificando a gestão e a escalabilidade das interações, utilizando URLs para identificar recursos e suportando operações padrão do HTTP (GET, POST, PUT, DELETE) para a manipulação desses recursos (JONSSON; QVARNSTRÖM, 2022). Um dos principais atrativos das APIs RESTful é a sua eficiência e facilidade de integração, impulsionadas pela simplicidade do JSON, que, por sua estrutura leve e legível, facilita a comunicação entre sistemas distintos e contribui para uma integração fluida em diversas aplicações e serviços web (BITTENCOURT, 2021).

2.2.1 Arquitetura REST

A arquitetura REST (Representational State Transfer) é um estilo arquitetônico inovador criado por Roy Fielding em sua tese de doutorado em 2000 (FIELDING, 2000). REST não se configura como um protocolo ou padrão rígido, mas sim como um conjunto de diretrizes e princípios destinados a orientar o desenvolvimento de sistemas de software distribuídos, o que faz com que essa abordagem seja amplamente adotada na construção de APIs modernas devido à sua flexibilidade e eficiência (GOLMOHAMMADI; ZHANG; ARCURI, 2023).

A essência da arquitetura REST está na sua simplicidade e na maneira como define a comunicação entre sistemas de forma escalável e de alta performance. Operando sobre o protocolo HTTP e enfatizando a separação entre cliente e servidor, a arquitetura permite que ambos evoluam de maneira independente, o que é fundamental para garantir a escalabilidade e a facilidade de manutenção dos sistemas distribuídos (EHSAN et al., 2022).

A arquitetura propõe um modelo de comunicação onde a ausência de estado (statelessness) é um princípio central (JONSSON; QVARNSTRÖM, 2022). Isso significa que cada requisição do cliente deve conter todas as informações necessárias para que o servidor possa processá-la, sem a necessidade de manter o estado da sessão entre as requisições (CARMO, 2023). Os princípios fundamentais da arquitetura REST são:

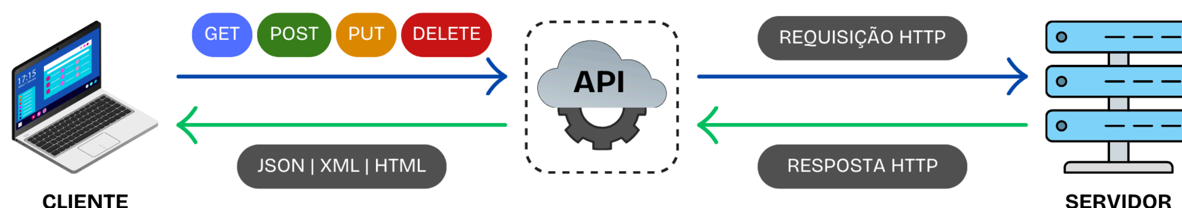
- Stateless (Sem Estado): Esse princípio exige que cada requisição do cliente contenha todas as informações necessárias para seu processamento, sem que o servidor mantenha dados sobre o estado entre as requisições. Isso melhora a escalabilidade e a eficiência, pois o servidor trata cada requisição de forma independente, sem precisar armazenar informações de sessão do cliente (TELLES, 2023).
- Client-Server (Cliente-Servidor): Estabelece uma clara separação entre o cliente, responsável pela interface do usuário e pela interação com o usuário, e o servidor, que gerencia o processamento de dados e a lógica de negócios. Essa divisão permite que ambos evoluam de forma independente e facilita a manutenção e escalabilidade do sistema, ao isolar as funções de interface das operações de backend (MAIOR, 2023).
- Cacheable (Cacheável): Permite que as respostas da API sejam armazenadas temporariamente no cliente, o que melhora significativamente o desempenho e

a escalabilidade do sistema ao reduzir a necessidade de buscar repetidamente as mesmas informações do servidor. Esse processo de cache não só promove a reutilização eficiente dos dados, mas também alivia a carga sobre o servidor, resultando em uma comunicação mais ágil e econômica (TELLES, 2023).

- Uniform Interface (Interface Uniforme): É um princípio que assegura uma comunicação padronizada entre cliente e servidor. Ela requer que recursos sejam identificados de maneira uniforme, que as representações de recursos contenham metadados para modificações e exclusões, e que mensagens sejam autodescritivas e incluam hiperlinks para recursos relacionados, facilitando a interoperabilidade e a compreensão das interações (MAIOR, 2023).
- HATEOAS (Hypermedia As The Engine Of Application State): É um princípio fundamental da arquitetura REST que permite que um servidor forneça ao cliente links dinâmicos para interações futuras, sem que o cliente precise conhecer previamente as URIs específicas. Em vez de exigir conhecimento antecipado sobre os recursos e endpoints, o servidor inclui hiperlinks nas respostas que guiam o cliente para as ações e recursos disponíveis, facilitando uma navegação mais intuitiva e flexível pela API (TELLES, 2023).
- Layered System (Sistema em Camadas): É um princípio que organiza o serviço em camadas distintas, como segurança, aplicação e lógica de negócios. Cada camada realiza funções específicas e interage de forma padronizada com as camadas adjacentes, mantendo essas camadas invisíveis para o cliente, promovendo modularidade. Além disso, a abordagem facilita a escalabilidade e a flexibilidade do sistema, já que novas funcionalidades podem ser adicionadas ou modificadas sem impactar outras camadas, simplificando a gestão e manutenção de sistemas complexos, uma vez que as responsabilidades são claramente delineadas e isoladas (TELLES, 2023).

A Figura 2 a seguir apresenta o modelo arquitetônico de uma API REST, evidenciando o fluxo de comunicação entre o usuário e o servidor. O diagrama ilustra como o cliente faz requisições ao servidor por meio da API usando métodos HTTP e como o servidor retorna os dados no formato adequado, como JSON, XML ou HTML, conforme as especificações da solicitação ou as configurações do servidor.

Figura 2 - Modelo arquitetural REST.



Fonte: Adaptada de 'Modelo arquitetural de uma API REST' por CARMO (2023).

2.2.2 Protocolo HTTP

O *Hypertext Transfer Protocol* (HTTP) é o protocolo fundamental da web, responsável por definir as regras de formatação e transmissão de mensagens entre clientes e servidores. Projetado inicialmente para a comunicação entre navegadores e servidores web, o HTTP também se aplica a uma ampla gama de outros propósitos. Operando na camada de aplicação do modelo OSI, o protocolo adota um modelo cliente-servidor clássico onde um cliente, como um navegador, estabelece uma conexão, envia uma requisição e aguarda uma resposta do servidor, que é 'stateless', não retendo informações sobre requisições anteriores e garantindo que cada requisição seja tratada de forma independente (MOZILLA, 2024).

A estrutura das mensagens HTTP é baseada em texto e se divide em requisições, enviadas pelo cliente para solicitar um recurso ou ação específica, e respostas, que são retornadas pelo servidor após processar essas requisições (FIELDING et al., 1999). Essa comunicação não se limita apenas a documentos HTML; o HTTP pode solicitar e entregar diversos tipos de recursos, como imagens, vídeos e scripts, e é a base para APIs RESTful, facilitando a comunicação entre diferentes sistemas e aplicações através da web (GOLMOHAMMADI; ZHANG; ARCURI, 2023).

Para compreender o funcionamento do HTTP, é crucial entender os diferentes métodos que ele utiliza para operar, definindo a natureza e o propósito das requisições, desde a simples obtenção de informações até a modificação de recursos. O HTTP também inclui métodos principais como GET e POST, além de códigos de status, que indicam o resultado das requisições e ajudam a interpretar a resposta do servidor.

2.2.3 Métodos HTTP

Os métodos HTTP são ações específicas que um cliente pode solicitar a um servidor para realizar operações sobre recursos identificados por URLs, onde cada método define um tipo de operação a ser executada e possui uma semântica distinta, permitindo que as APIs,

como as APIs RESTful, manipulem recursos de maneira padronizada. Esses métodos são essenciais para a comunicação entre cliente e servidor, pois indicam a intenção da requisição e determinam a forma como o servidor deve responder (FIELDING et al., 1999). Na Tabela 1, são apresentados os principais métodos HTTP utilizados, detalhando suas funções, usos comuns e características, como a idempotência, que descreve se múltiplas requisições idênticas têm o mesmo efeito que uma única requisição.

Tabela 1 - Principais métodos HTTP.

Método	Ação	Descrição	Idempotência
GET	Ler	Solicita a representação de recursos	Sim
POST	Criar	Enviar dados para criar um novo recurso	Não
PUT	Atualizar/Criar	Substitui ou cria um recurso com os dados fornecidos	Sim
DELETE	Deletar	Remove um recurso específico	Sim
PATCH	Atualizar parcialmente	Aplica modificações parciais em um recurso	Não

Fonte: Elaborada pela autora (2024).

2.2.4 Códigos de Status HTTP

Os códigos de status HTTP são códigos numéricos retornados pelo servidor para indicar o resultado de uma solicitação, que inclui um código de três dígitos que informa se a solicitação foi bem-sucedida ou se ocorreu algum tipo de erro. Este código fornece uma indicação clara do resultado da operação solicitada, permitindo que o cliente entenda rapidamente o status da requisição, pois fornecem informações detalhadas sobre o sucesso, falhas ou a necessidade de redirecionamentos em relação à requisição feita (FIELDING et al., 1999). As categorias de códigos de status são divididas em cinco grupos principais, cada uma refletindo um tipo diferente de resposta do servidor, como apresentado na Tabela 2.

Tabela 2 - Categorias de códigos de status HTTP.

Código	Categoria	Descrição
1xx	Informativo	Indica que a requisição está sendo processada

2xx	Sucesso	Confirma que a requisição foi bem-sucedida
3xx	Redirecionamento	Indica que ações adicionais são necessárias para completar a requisição
4xx	Erro do Cliente	Indica problemas com a requisição enviada pelo cliente
5xx	Erro do Servidor	Indica que o servidor encontrou um erro ao processar a requisição

Fonte: Elaborada pela autora (2024).

Cada categoria de códigos de status HTTP é subdividida em intervalos de 100, oferecendo informações detalhadas sobre o resultado da requisição, como por exemplo, a categoria 2xx, abrange códigos de 200 a 299, onde o código 200 (Ok) indica que a solicitação foi bem-sucedida. Da mesma forma, a categoria 4xx vai de 400 a 499 e reflete problemas com a requisição enviada pelo cliente, como o código 404 (Not Found) para recursos não encontrados (FIELDING et al., 1999). Essas subcategorias permitem uma compreensão mais precisa da resposta do servidor e facilitam a resolução de problemas na comunicação entre cliente e servidor.

2.2.5 Endpoints

Os endpoints (ou terminais, em português) são os pontos de contato que uma API disponibiliza para permitir a comunicação com outros sistemas. Estes pontos de contato são representados por URLs (Uniform Resource Locators), que identificam recursos específicos no servidor (BITTENCOURT, 2021). Cada URL é composta por diferentes partes, que juntas definem o caminho para acessar um recurso ou executar uma ação. Uma URL típica de uma API web possui uma estrutura básica, como ilustrado na Figura 3.

Figura 3 - Estrutura de uma URL.

https://exemplo.com/recurso/1

Fonte: Elaborada pela autora (2024).

A estrutura básica de uma URL inclui:

- Domínio Base: *https://exemplo.com* é o domínio base da API, que é o endereço principal e indica onde a API está hospedada.

- Caminho para o Recurso: Após o domínio base, a URL inclui um caminho, como */recurso*, que especifica o recurso ou a coleção de recursos a ser acessada.
- Identificador Único (Opcional): Em alguns casos, a URL pode incluir um identificador único para um recurso específico. Por exemplo, */recurso/1* aponta para um recurso com ID 1.

Os endpoints, independentemente da arquitetura da API, são sempre os pontos na borda da API que recebem as requisições, desempenhando um papel crucial na comunicação entre clientes e servidores, garantindo que as operações sejam realizadas de forma organizada e segura. Além disso, as respostas de muitos endpoints são frequentemente formatadas em JSON, um formato leve e amplamente utilizado para a troca de dados, que facilita a comunicação eficiente entre sistemas.

2.2.6 Representação de Dados com JSON

O *JavaScript Object Notation* (JSON) é um formato leve e de fácil leitura, amplamente utilizado para a troca de dados entre servidores e clientes, especialmente em APIs web. Embora o formato JSON derive da notação de objetos JavaScript, ele é um texto que pode ser facilmente trafegado entre aplicações e plataformas diversas, utilizando protocolos como HTTP (LEITE, 2023).

No contexto das APIs RESTful, o JSON se destaca como o formato de troca de dados predominante, graças à sua capacidade de representar dados estruturados de forma clara e compacta (BITTENCOURT, 2021). A Figura 4 exemplifica a estrutura de um objeto JSON, destacando a forma como os dados são representados e organizados.

Figura 4 - Representação de um JSON.


```
{
  "usuarios": [
    {
      "id": 1,
      "nome": "Fulano",
      "email": "fulano@example.com"
    }
  ]
}
```


Fonte: Elaborada pela autora (2024).

2.3 NODE.JS

O Node.js é uma plataforma de código aberto e multiplataforma desenvolvida em 2009 por Ryan Dahl, que permite a execução de JavaScript no lado do servidor usando o motor V8 do Google Chrome (DEMIR; NILSSON, 2023). Essa tecnologia se destaca por seu modelo de processamento assíncrono e não-bloqueante, o que a torna particularmente eficiente para aplicações que realizam muitas operações de entrada/saída (I/O) e manipulação de arquivos, o que o torna ideal para aplicações que exigem alta performance e escalabilidade, como servidores de APIs RESTful. (PEREIRA, 2014; SABO, 2020).

Entre suas principais características, destaca-se o seu modelo I/O não-bloqueante, pois em vez de criar uma nova thread para cada solicitação, o Node.js utiliza uma única thread para processar requisições de maneira assíncrona. Isso significa que, enquanto aguarda a conclusão de operações como acesso a bancos de dados ou leitura de arquivos, o Node.js continua a processar outras tarefas (PEREIRA, 2014).

Além disso, a capacidade de gerenciar milhares de conexões simultâneas de forma eficiente e escalável é uma grande vantagem. Outro ponto forte é sua integração com gerenciadores de pacotes, como o NPM (Node Package Manager), que oferece um vasto repositório de módulos e bibliotecas, facilitando o desenvolvimento e a adição de novas funcionalidades, simplificando o processo de construção e manutenção de projetos. Sua escalabilidade horizontal e a ampla adoção por grandes empresas e startups reforçam sua eficácia e popularidade (CARMO, 2023).

2.3.1 O Motor V8

O motor V8, criado pela Google, é uma peça fundamental para o desempenho eficiente do Node.js, sendo escrito em C++ e disponibilizado como um projeto de código aberto, foi desenvolvido para executar o código JavaScript com alta performance (KRYLOV et al., 2020). Uma das suas principais características é a capacidade de compilar JavaScript diretamente em código de máquina nativo por meio de técnicas de compilação *Just in Time* (JIT), o que elimina a necessidade de interpretação em tempo real e proporciona uma execução mais rápida e eficiente (CARMO, 2023).

Além de sua função na compilação, o V8 também se destaca no gerenciamento de memória, sendo responsável pela alocação e a coleta de lixo, que é um processo automatizado que remove da memória objetos que não são mais necessários, o que é vital para prevenir vazamentos de memória e manter a estabilidade das aplicações. Em cenários que demandam

alta performance e lidam com cargas intensivas de processamento, o gerenciamento eficiente de memória pelo V8 é um fator crucial para o bom funcionamento do Node.js (KRYLOV et al., 2020).

2.3.2 Framework NestJS

O NestJS é um framework para o desenvolvimento de aplicações no ambiente Node.js, sendo uma escolha popular para o desenvolvimento de APIs RESTful. Desenvolvido com TypeScript, o NestJS se destaca por sua arquitetura modular, facilitada pela sua base sobre a arquitetura de injeção de dependências; e pela capacidade de suportar aplicações escaláveis e de fácil manutenção (NESTJS, [s.d.]). A integração do NestJS com o Node.js permite aproveitar os benefícios da arquitetura não-bloqueante do Node.js, ao mesmo tempo que proporciona uma base sólida e extensível para o desenvolvimento de APIs e serviços (SABO, 2020).

Integra a filosofia de desenvolvimento do Node.js, aproveitando seu modelo de I/O não-bloqueante, permitindo que aplicações construídas com NestJS possam lidar com um grande volume de requisições simultâneas de forma eficiente e escalável. O NestJS funciona como uma camada de abstração em cima de frameworks de servidor existentes, como o Express.js, embora também suporte outras alternativas se necessário (SABO, 2020).

A arquitetura do NestJS é projetada para oferecer uma estrutura clara e organizada, essencial para o desenvolvimento de aplicações robustas e escaláveis, sendo composta por três principais componentes:

- Modularidade: A estrutura do NestJS é fortemente modular, dividindo a aplicação em módulos distintos que agrupam funcionalidades relacionadas, cada módulo é responsável por um conjunto específico de funcionalidades e pode exportar serviços e controladores para outros módulos. Esta abordagem modular não apenas promove uma organização eficiente do código, mas também facilita a manutenção e escalabilidade da aplicação, permitindo a adição e remoção de funcionalidades com impacto mínimo no restante do sistema (SABO, 2020; NESTJS, [s.d.]).
- Controladores: São componentes que lidam com as requisições HTTP recebidas e retornam as respostas apropriadas, servindo como intermediários entre a camada de apresentação da aplicação e a lógica de negócios. Os

controladores recebem os dados das requisições, processam eles conforme necessário e delegam a lógica de negócios para os serviços apropriados (SABO, 2020; NESTJS, [s.d.]).

- Serviços: Responsáveis por encapsular a lógica de negócios da aplicação, sendo esses componentes injetados nos controladores e módulos, permitindo a centralização e reutilização da lógica de negócios. A injeção de dependências ajuda a manter o código modular e organizado, facilitando o gerenciamento e a testabilidade da aplicação (SABO, 2020; NESTJS, [s.d.]).

2.4 .NET

O .NET é uma plataforma de desenvolvimento de código aberto, multiplataforma e gratuita criada pela Microsoft e pela .NET Foundation, sendo lançada como uma solução para criar uma ampla gama de aplicativos, incluindo web, desktop e mobile, suportando várias linguagens de programação, como C#, F# e Visual Basic (DALBARD; ISACSON, 2021). A plataforma é modular e composta por um conjunto robusto de bibliotecas e ferramentas que facilitam o desenvolvimento de aplicações complexas.

Uma das características do .NET é a sua capacidade de operar em diferentes sistemas operacionais, o que inclui Windows, Linux e macOS. Através do uso de pacotes NuGet, desenvolvedores podem integrar bibliotecas de código aberto em seus projetos, aproveitando um vasto ecossistema de recursos disponíveis (DALBARD; ISACSON, 2021).

2.4.1 Framework ASP.NET Core

O ASP.NET Core é um framework web de código aberto que se baseia na plataforma .NET, lançado pela Microsoft em 2016, sendo sucessor do ASP.NET tradicional, oferecendo melhorias significativas em modularidade, flexibilidade e suporte multiplataforma (MAIOR, 2023). Diferente do ASP.NET original, que era restrito ao Windows, o ASP.NET Core permite a execução em Windows, Linux, macOS e containers Docker (KARLSSON, 2021). Entre suas principais características estão:

- Desempenho: O ASP.NET Core é conhecido por sua alta performance, oferecendo suporte a técnicas de otimização como o pipeline de middleware e a implementação de serviços eficientes (MICROSOFT, 2024).
- Modularidade: Sua arquitetura modular permite que desenvolvedores escolham

e integrem apenas os componentes necessários para suas aplicações, facilitando a personalização e a manutenção (MICROSOFT, 2024).

- Suporte a Dependency Injection: O framework tem suporte nativo para injeção de dependências, o que promove uma estrutura de código limpa e testável (MICROSOFT, 2024).

O ASP.NET Core é projetado para operar tanto de forma síncrona quanto assíncrona, utilizando uma abordagem *multi-threaded* para gerenciar múltiplas solicitações simultâneas. Em um ambiente ASP.NET Core, cada solicitação do cliente pode ser tratada por uma thread dedicada, o que contribui para uma melhor performance e escalabilidade das aplicações web. O framework emprega um pool de threads para lidar com o aumento da carga de trabalho de maneira eficiente, permitindo que o sistema gerencie um grande número de requisições simultâneas sem sobrecarregar o sistema (KARLSSON, 2021).

2.5 GO

Go, também conhecido como Golang, é uma linguagem de programação desenvolvida pelo Google que combina simplicidade, eficiência e suporte robusto para a concorrência. Desde seu lançamento como código aberto em 2009, Go se destacou pela facilidade de uso e pela capacidade de lidar com aplicações escaláveis e de alto desempenho (GO, 2019; SRIVASTAVA, 2023). Como características principais, podemos citar:

- Simplicidade e Clareza: A sintaxe do Go é direta, o que facilita o aprendizado e a leitura do código, tornando a linguagem acessível tanto para iniciantes quanto para programadores experientes (SRIVASTAVA, 2023).
- Suporte à Concorrência: Com recursos como *goroutines* e canais, Go permite a execução simultânea de múltiplas tarefas, o que é ideal para aplicações que exigem alta performance em processamento paralelo (GO, 2019).
- Gerenciamento de Memória: A linguagem inclui um coletor de lixo, o que reduz a carga sobre os desenvolvedores em relação à alocação e liberação de memória (SRIVASTAVA, 2023).
- Compilação Rápida: Os tempos de compilação curtos possibilitam iterações rápidas, melhorando a eficiência do desenvolvimento (GO, 2019).
- Portabilidade: Aplicações escritas em Go podem ser executadas em diversos

sistemas operacionais, facilitando o desenvolvimento multiplataforma (GO, 2019).

2.5.1 Framework Echo

O Echo é um framework para Go, projetado para alta performance e flexibilidade na criação de aplicações web e APIs. Suas principais características incluem:

- Roteamento Eficiente: Possui um roteador HTTP otimizado que prioriza rotas sem alocação dinâmica de memória, melhorando a performance e o uso de recursos (ECHO, [s.d.]).
- Escalabilidade: Facilita a construção de APIs RESTful escaláveis, organizando endpoints em grupos lógicos para uma gestão simplificada (ECHO, [s.d.]).
- TLS Automático: Gerencia automaticamente a instalação de certificados TLS com Let's Encrypt, simplificando a configuração de conexões seguras (ECHO, [s.d.]).
- Suporte a HTTP/2: Adota o protocolo HTTP/2 para otimizar a velocidade e a eficiência da comunicação entre servidor e cliente (ECHO, [s.d.]).
- Middleware: Oferece um conjunto de funções middleware integradas e permite a criação de middleware personalizado para ajustar a aplicação conforme necessário (ECHO, [s.d.]).
- Binding e Renderização de Dados: Facilita a vinculação de dados de requisições e a renderização de respostas em vários formatos, como JSON, XML e HTML (ECHO, [s.d.]).
- Templates: Suporta a renderização de templates com qualquer motor de template, permitindo a criação de conteúdo dinâmico (ECHO, [s.d.]).
- Extensibilidade: Permite personalizar o tratamento de erros e adicionar middleware ou plugins personalizados, adaptando o framework às necessidades específicas do projeto (ECHO, [s.d.]).

2.6 FERRAMENTA DE TESTE DE DESEMPENHO

No desenvolvimento de software, especialmente em aplicações web e APIs, garantir o desempenho adequado é essencial para uma boa experiência do usuário e para a eficiência operacional, e utilizar ferramentas de teste de desempenho ajuda a identificar gargalos, medir

a capacidade de resposta e a escalabilidade (CHIMUCO; BARBOSA, 2024). Em ambientes como Node.js e .NET, assim como em Go, onde o desempenho e a escalabilidade são essenciais, o uso de ferramentas de desempenho, como o k6, é crucial para monitorar e otimizar a performance das aplicações.

2.6.1 k6

O k6 é uma ferramenta de código aberto desenvolvida para a execução de testes de carga e desempenho, criada com a finalidade de facilitar a avaliação da performance de aplicações e APIs, foi construído na linguagem Go, oferecendo uma solução eficiente com foco em um baixo consumo de recursos (GRAFANA, [s.d.]). A ferramenta utiliza JavaScript para a criação de scripts, o que permite aos desenvolvedores uma experiência familiar e prática na configuração dos testes (MORAES, 2023).

O k6 é projetado para ser uma solução acessível e altamente integrada ao processo de desenvolvimento contínuo, facilitando a inclusão de testes de desempenho em pipelines de CI/CD. Sua interface de linha de comando simplificada e seu suporte para recursos avançados, como verificação e definição de limites personalizados, tornam o k6 uma escolha prática para desenvolvedores que buscam automatizar e otimizar a análise de carga e desempenho (MORAES, 2023).

2.7 TRABALHOS RELACIONADOS

Diversos estudos têm explorado a comparação de arquiteturas e ferramentas para o desenvolvimento e otimização de APIs, com foco em aspectos como desempenho, escalabilidade e facilidade de uso. Maso (2024) realizou uma análise comparativa entre as arquiteturas de APIs REST, GraphQL e gRPC, destacando as diferenças em termos de desempenho, utilização de recursos e adequação a diferentes contextos de aplicação. O estudo de Maso demonstrou que, embora a arquitetura REST ainda seja amplamente utilizada, alternativas como GraphQL e gRPC oferecem benefícios específicos em cenários que requerem maior eficiência e menor sobrecarga de dados.

Carmo (2023) conduziu um estudo comparativo entre três das principais tecnologias de back-end: Node.js, Django REST Framework e ASP.NET Core, com o objetivo de avaliar o desempenho, a facilidade de uso e a compatibilidade de cada tecnologia em diferentes cenários de desenvolvimento de APIs. O estudo envolveu a implementação de API em cada uma das tecnologias e a realização de testes de desempenho sob diferentes condições de

carga. Os resultados mostraram que, embora o Node.js e o Django REST Framework apresentem baixo consumo de recursos, o ASP.NET Core se destacou com o menor tempo de resposta.

Jonsson e Qvarnström (2022) compararam o desempenho de três frameworks populares para APIs REST: Express.js, Flask e ASP.NET Core, focando em métricas como throughput, tempo de resposta e uso de recursos. Usando o JMeter para testar diferentes condições de carga, descobriram que o ASP.NET Core teve o melhor desempenho em tempo de resposta e throughput, especialmente sob cargas altas. O Flask ficou atrás dos outros frameworks, e o Express.js foi eficiente com baixa carga, mas superado pelo ASP.NET Core em cargas maiores.

Adicionalmente, Maior (2023) investigou o desempenho de vários frameworks para APIs REST, usando um modelo de e-commerce para comparar Spring Boot, Express.js, ASP.NET Core e Django, utilizando a ferramenta JMeter para avaliar o tempo de resposta, vazão de requisições e taxa de erros em condições de estresse. Os resultados indicaram que ASP.NET Core e Spring Boot foram superiores em processamento e concorrência, Express.js se destacou pela simplicidade e eficiência em baixa carga, enquanto o Django enfrentou desafios com múltiplas requisições simultâneas, embora possa ser otimizado com ajustes específicos.

Apesar dos avanços na comparação de desempenho entre diferentes tecnologias, ainda não há estudos na literatura que comparem diretamente APIs em Node.js, .NET e Go em termos de desempenho. Este estudo se propõe a preencher essa lacuna, focando especificamente na métrica de tempo de resposta e considerando os principais métodos HTTP. Utilizando o k6, serão avaliados vários cenários para fornecer uma visão detalhada do desempenho dessas três tecnologias em diferentes condições de carga.

3. METODOLOGIA

Este capítulo apresenta a caracterização da pesquisa, bem como os procedimentos e técnicas utilizadas para coleta e análise dos dados, além do detalhamento do método de pesquisa realizado para atingir os objetivos deste trabalho.

3.1 CARACTERIZAÇÃO DA PESQUISA

A pesquisa, segundo Perovano (2016), é essencial para a produção e difusão do conhecimento, construção de conceitos e teorias, e inovação em ciência e tecnologia. Ela permite o desenvolvimento de novos conceitos e teorias, assim como a revisão de estruturas teóricas existentes. Diferentes métodos são aplicados na pesquisa científica para explorar as várias modalidades de investigação (SEVERINO, 2017). A abordagem metodológica adotada neste estudo, sintetizada no Quadro 1, ilustra as técnicas e métodos utilizados para a realização da pesquisa.

Quadro 1 - Quadro metodológico da pesquisa.

Finalidade	Aplicada
Natureza	Empírica
Abordagem	Quantitativa
Objetivos	Descritiva
Estratégia	Experimento Controlado
Método científico	Dedutivo e Indutivo
Procedimentos técnicos	Experimento controlado, coleta de dados de desempenho, análise estatística.
Procedimentos para coleta de dados	Medição de métricas de desempenho (tempo de resposta), registro de dados experimentais, análise estatística.

Fonte: Elaborada pela autora (2024).

3.1.1 Quanto à Finalidade

Quanto à finalidade do estudo, a pesquisa é aplicada, pois busca comparar o desempenho de APIs RESTful desenvolvidas em Node.js, .NET e Go. A pesquisa aplicada tem um viés prático, focando na resolução de problemas específicos e no atendimento de

demandas organizacionais. Segundo Perovano (2016), ela se concentra em resolver problemas concretos através de esforços integrados e específicos. Este estudo visa fornecer *insights* diretos para desenvolvedores e profissionais de TI sobre o desempenho de APIs em diferentes tecnologias em termos de tempo de resposta, ajudando na tomada de decisões informadas e na melhoria das práticas tecnológicas.

3.1.2 Quanto à Natureza

A pesquisa possui natureza empírica, uma vez que envolve a realização de experimentos controlados para coletar dados observáveis sobre o desempenho de APIs RESTful desenvolvidas em Node.js, .NET e Go em diferentes cenários. A coleta de dados empíricos é essencial para entender como diferentes tecnologias influenciam o desempenho das APIs na prática.

Segundo Yin (2004), abordagens empíricas como estudos de caso e experimentos controlados são fundamentais para validar e aplicar teorias na prática. Além disso, teorias sobre o desempenho de sistemas de informação, discutidas pela comunidade acadêmica, oferecem uma estrutura teórica robusta para compreender como fatores técnicos impactam o desempenho das tecnologias da informação.

3.1.3 Quanto à Abordagem

A abordagem adotada para o problema é quantitativa, o que permite quantificar e comparar métricas de desempenho das APIs RESTful desenvolvidas em Node.js, .NET e Go, como tempo médio de resposta das requisições. Esta escolha metodológica é fundamentada na necessidade de uma análise objetiva e sistemática das diferenças de desempenho entre as soluções estudadas, alinhando-se com a prática recomendada por autores como Yin (2004), que destacam como a abordagem quantitativa proporciona *insights* replicáveis e fundamentados sobre fenômenos complexos.

3.1.4 Quanto ao Objetivo

Em relação ao objetivo, a pesquisa é descritiva, cujo foco principal é descrever o desempenho das APIs RESTful desenvolvidas em Node.js, .NET e Go sob diferentes condições de carga e utilizando os principais métodos HTTP. Essa escolha metodológica permite uma análise minuciosa das características e funcionalidades de cada abordagem, oferecendo uma visão abrangente das capacidades e limitações das tecnologias avaliadas.

Conforme descrito por Trochim e Donnelly (2020), estudos descritivos são elaborados principalmente para fornecer uma descrição de eventos ou fenômenos que estão ocorrendo ou que existem. Analogamente, ao buscar apenas descrever a variação no desempenho das APIs, esta pesquisa se alinha com a natureza descritiva, proporcionando uma compreensão detalhada das diferenças observadas na utilização de tecnologias diferentes.

3.1.5 Quanto a Estratégia

A estratégia de pesquisa selecionada para este estudo será a de experimento controlado. Nessa abordagem, condições específicas serão estabelecidas e manipuladas para avaliar o desempenho das APIs desenvolvidas em Node.js, .NET e Go. Essa metodologia permite isolar variáveis-chave e controlar possíveis influências externas, garantindo assim a validade interna dos resultados obtidos.

A escolha por um experimento controlado é fundamentada na necessidade de realizar comparações diretas e objetivas entre as diferentes soluções avaliadas. Segundo Trochim e Donnelly (2020), experimentos controlados são particularmente eficazes para testar hipóteses e investigar relações de causa e efeito, proporcionando uma base sólida para a análise comparativa de desempenho tecnológico.

3.1.6 Quanto ao Método Científico

Trochim e Donnelly (2020) explicam que o método dedutivo procede do mais geral para o mais específico, começando com teorias amplas que são refinadas em hipóteses específicas testáveis. Em contrapartida, o método indutivo parte de observações específicas para desenvolver generalizações e teorias mais abrangentes, explorando padrões identificados nos dados para formular novas teorias.

Para esta pesquisa, foi adotado tanto o método dedutivo quanto o indutivo. O método dedutivo será utilizado para estabelecer hipóteses específicas com base em teorias existentes sobre o impacto das tecnologias Node.js, .NET e Go no desempenho das APIs. A partir dessas hipóteses, serão realizados experimentos controlados para testar empiricamente as expectativas teóricas.

Por outro lado, o método indutivo será empregado para analisar os dados coletados durante os experimentos. A partir das observações concretas sobre o desempenho das APIs desenvolvidas com Node.js, .NET e Go, serão formuladas generalizações e conclusões gerais. Este método permite extrair conclusões amplas e identificar padrões com base nos resultados

empíricos obtidos.

A combinação desses métodos proporcionará uma abordagem abrangente para explorar e interpretar as complexidades do desempenho das APIs desenvolvidas em Node.js, .NET e Go.

3.1.7 Quanto aos Procedimentos Técnicos

Os procedimentos técnicos incluirão a realização de experimentos controlados utilizando cada API desenvolvida (Node.js, .NET e Go), a coleta de dados de desempenho, como o tempo de resposta das requisições, e a subsequente análise estatística dos resultados obtidos. Para a medição precisa e eficiente das métricas de desempenho, será utilizado o k6, uma ferramenta reconhecida pela sua capacidade de simular cargas de trabalho e monitorar o desempenho de sistemas distribuídos.

Esses passos são fundamentais para assegurar a validade e a confiabilidade dos dados coletados, conforme destacado por Trochim e Donnelly (2020), proporcionando uma análise objetiva sobre o impacto de cada tecnologia no desempenho das APIs.

3.1.8 Quanto aos Procedimentos para coleta de dados

Os procedimentos para coleta de dados envolvem a utilização do k6 para realizar medições precisas de métricas de desempenho durante a execução dos experimentos. Serão coletados os dados de tempo de resposta das requisições para avaliar o desempenho das APIs desenvolvidas nas tecnologias Node.js, .NET e Go.

Durante a execução dos testes, os dados experimentais serão registrados sistematicamente para garantir a integridade e precisão das informações coletadas. Isso inclui a captura detalhada das configurações de teste, como número de usuários, configurações de carga e parâmetros de teste específicos de cada API.

Posteriormente, os dados serão submetidos a uma análise estatística robusta. Essa análise visa comparar os resultados obtidos entre os diferentes gerenciadores de pacotes, utilizando métodos como médias e gráficos de distribuição. Essas técnicas proporcionam *insights* precisos de como cada tecnologia impacta o desempenho das APIs na prática.

3.2 MÉTODO DE PESQUISA

Um método de pesquisa é uma sequência estruturada de passos essenciais para alcançar os objetivos propostos de maneira convincente. Seguir as etapas delineadas no

método é crucial para garantir resultados robustos e válidos. A definição clara do método de pesquisa orienta o pesquisador na investigação e na resposta ao problema de pesquisa, não apenas facilitando a validade da pesquisa, mas também fortalecendo sua credibilidade (MARCONI; LAKATOS, 2017). A próxima seção detalha o método específico adotado neste estudo.

3.2.1 Método Experimental

O método experimental envolve a manipulação de variáveis específicas em condições controladas para observar os efeitos produzidos nos objetos de estudo (PROVDANOV; FREITAS, 2013). Em outras palavras, é um conjunto de procedimentos através dos quais um pesquisador realiza um experimento, manipulando diversas variáveis que se relacionam com o objeto de estudo (CAMPBELL; STANLEY, 2015).

Conforme Trochim e Donnelly (2020), o método experimental permite ao pesquisador controlar e/ou manipular intencionalmente uma ou mais variáveis para observar os efeitos sobre uma ou mais variáveis dependentes. Este controle é essencial para criar condições específicas que permitem observar como uma variável dependente se comporta em diferentes cenários experimentais, facilitando a identificação de relações de causa e efeito entre as variáveis e a formulação de conclusões robustas (PROVDANOV; FREITAS, 2013).

Em suma, o método experimental é fundamental para investigar as interações de variáveis em situações específicas, contribuindo significativamente para o avanço do conhecimento científico ao permitir testar hipóteses de maneira controlada e sistemática.

3.3 EXPERIMENTO DA PESQUISA

O objetivo do experimento controlado nesta pesquisa foi analisar o desempenho de APIs desenvolvidas nas tecnologias Node.js, .NET e Go. Para a ocorrência do experimento, o desempenho foi testado colocando o servidor sob diferentes cargas e medindo o tempo de resposta das solicitações. O script do k6 foi configurado com as solicitações enviadas e consequentemente, captura das respostas nos diferentes cenários e casos de teste detalhados no Capítulo 4 deste trabalho.

3.3.1 Ambiente Controlado

Neste estudo, o ambiente controlado para a execução do experimento envolveu a utilização da ferramenta k6 em um computador com as seguintes especificações técnicas:

- **Modelo do Computador:** Dell Inspiron 15 3520
- **Processador:** 11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz, 2419 Mhz, 4 Núcleo(s), 8 Processadores Lógicos.
- **Memória RAM:** 16 GB.
- **Armazenamento:** 512GB SSD.
- **Sistema Operacional:** Microsoft Windows 11 Home.

Essas especificações foram determinadas para garantir um ambiente controlado e estável durante a execução dos testes de desempenho das APIs nas tecnologias Node.js, .NET e Go.

3.3.2 Métrica analisada

Para avaliar o desempenho, a métrica observada nos experimentos para posterior análise, foi o tempo de resposta. Seguindo a definição de Dhalla (2021), o tempo de resposta refere-se ao intervalo de tempo em milissegundos necessário para que um serviço web atenda a uma solicitação feita pelo cliente.

3.4 MÉTODOS E TÉCNICAS DE COLETA E ANÁLISE DE DADOS

Para a coleta e análise de dados deste estudo, foi utilizada a observação sistemática. Os dados coletados consistem na resposta de cada solicitação enviada ao servidor em cada execução dos cenários e casos configurados nos scripts do k6, detalhados no Capítulo 4.

3.4.1 Observação sistemática

A observação sistemática possibilita o registro organizado e preciso das variáveis de interesse, como o tempo de resposta das APIs, sob diferentes condições de carga e com as tecnologias Node.js, .NET e Go. Esta metodologia assegura a objetividade na coleta de dados, utilizando protocolos estruturados que facilitam a categorização e análise posterior das informações obtidas (COZBY, 2012; GRAY, 2012).

Dessa forma, desempenha um papel fundamental na validação dos experimentos realizados, fornecendo uma base sólida para as conclusões e recomendações derivadas deste estudo sobre o desempenho das APIs nas diferentes tecnologias analisadas.

4. IMPLEMENTAÇÃO

Neste capítulo, será descrito o processo de desenvolvimento e testes das APIs. Sendo dividida em quatro seções principais: desenvolvimento das APIs, com ênfase na arquitetura e estrutura do projeto; descrição dos cenários de teste; construção dos scripts de teste; e execução dos testes com a ferramenta k6.

4.1 DESENVOLVIMENTO DAS APIs

O desenvolvimento das APIs para esta pesquisa envolveu a criação de três versões semelhantes, cada uma implementada em uma linguagem e framework diferentes: Node.js (20.10.0) com NestJS (10.4.5), ASP.NET Core (8.0.1), e Go (1.22.6) com Echo (4.12.0), conforme apresentados no Capítulo 2 deste trabalho.

Cada API seguiu um padrão consistente de arquitetura, com componentes claramente separados em *controllers*, *services* (ou *handlers* no caso do Go) e *repositories*. No Node.js, o Prisma ORM (5.16.1) foi empregado para simplificar a interação com o banco de dados PostgreSQL, enquanto o Entity Framework Core (8.0.8) foi utilizado no ASP.NET Core, e a biblioteca GORM (1.25.12) foi adotada para o Go, fornecendo assim, uma camada de abstração eficiente para a manipulação de dados.

Assim, enquanto a lógica de negócio e a estrutura das APIs permanecem constantes entre as três implementações, as diferenças no ambiente de execução e frameworks permitirão uma análise comparativa detalhada do impacto no desempenho de cada uma.

4.1.1 Descrição e Funcionalidades das APIs

As APIs desenvolvidas para esta pesquisa tem como objetivo fornecer uma interface para a gestão de dados de livros e gêneros de livros. Cada API implementa um conjunto de operações CRUD (Create, Read, Update, Delete) para as entidades de livro e gênero de livro, permitindo a criação, leitura, atualização e exclusão de registros no banco de dados. A seguir, está um breve resumo das funcionalidades principais:

- CRUD de Livros: Permite a criação de novos livros, recuperação de informações de livros existentes, atualização de dados dos livros e remoção de livros do sistema.
- CRUD de Gêneros de Livros: Oferece funcionalidades semelhantes de livros,

incluindo a adição de novos gêneros livros, visualização de detalhes dos gêneros de livros, atualização das informações e exclusão de gêneros de livros do sistema.

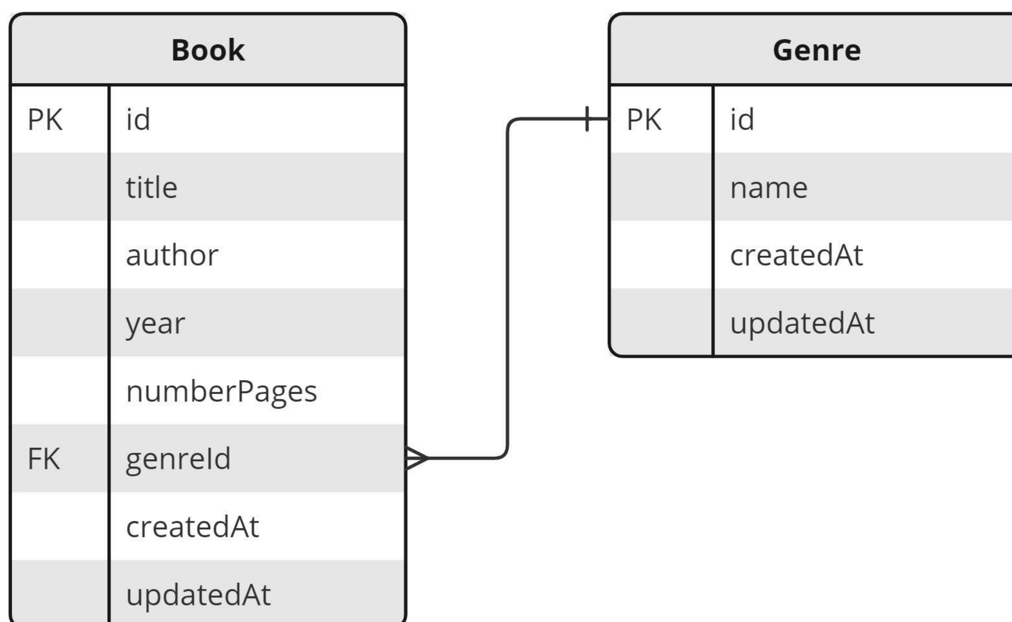
4.1.2 Banco de Dados

O banco de dados utilizado para todas as APIs é o PostgreSQL, um sistema de gerenciamento de banco de dados relacional amplamente utilizado, conhecido por sua robustez e suporte a operações complexas. A interação com o PostgreSQL é facilitada pelos ORMs e bibliotecas mencionados anteriormente para cada tecnologia, que oferecem uma abstração eficiente para a manipulação dos dados.

4.1.2.1 Diagrama de Entidade-Relacionamento

O Diagrama de Entidade-Relacionamento (ER) é uma ferramenta fundamental para a modelagem de dados, permitindo a representação gráfica das entidades (tabelas) e dos relacionamentos entre elas, facilitando a visualização da organização dos dados e das conexões entre as diferentes tabelas no banco de dados. A Figura 5 ilustra a estrutura do banco de dados utilizado nesta pesquisa, oferecendo uma visão detalhada dos elementos e sua inter-relação.

Figura 5 - Diagrama de Entidade-Relacionamento.



Fonte: Elaborada pela autora (2024).

Analisando o diagrama, pode-se observar que:

- A tabela Book (livros) contém informações essenciais sobre cada livro, como título, autor, ano de publicação e número de páginas. Ela possui uma relação com a tabela Genre (gêneros) através do campo *genreId*, que serve como uma chave estrangeira (FK), indicando o gênero ao qual o livro pertence.
- A tabela Genre contém os dados relacionados aos diferentes gêneros disponíveis, como o nome do gênero.
- O relacionamento entre as duas tabelas é do tipo 1:N (um para muitos), o que significa que um gênero pode estar associado a vários livros, mas cada livro pertence a apenas um gênero.

Essa organização permite uma modelagem clara e eficiente dos dados, garantindo a integridade e o bom desempenho das consultas no banco de dados.

4.1.3 API em Node.js com NestJS

Nesta seção, é apresentada a implementação da API utilizando Node.js com NestJS, destacando suas principais características e a estrutura modular adotada.

4.1.3.1 Arquitetura da API

A arquitetura adotada no desenvolvimento da API seguiu o padrão de arquitetura modular do NestJS, que foi apresentada na seção 2.3.2 do Capítulo 2 desta pesquisa. Cada módulo é responsável por uma parte específica da aplicação, contendo em cada um seus Controllers, Services e Repositories, permitindo uma separação clara das responsabilidades.

4.1.3.1.1 Controllers

São responsáveis por receber as requisições HTTP e enviar as respostas, com cada controller se comunicando com os serviços para aplicar a lógica de negócios e garantir que os dados corretos sejam processados e retornados. A Figura 6 apresenta o exemplo do método POST, que é um dos métodos presentes no *Controller* de Livros, sendo este método responsável por processar requisições HTTP para criar novos registros de livros.

Figura 6 - Método POST no *Controller* de Livros.

```
@Post()
async create(@Body() createBookDto: CreateBookDto) {
  try {
    const book = await this.bookService.create(createBookDto);
    return { statusCode: HttpStatus.CREATED, book };
  } catch (error) {
    throw new HttpException(error.message, HttpStatus.INTERNAL_SERVER_ERROR);
  }
}
```

Fonte: Elaborada pela autora (2024).

No código apresentado, o método create é um manipulador de requisições POST que cria um novo livro, ao receber um objeto *createBookDto* no corpo da requisição e utilizando o *bookService* para processar a criação do livro. Se a operação for bem sucedida, o método retorna um status HTTP 201 (Created) junto com o objeto do livro criado. Em caso de erro, o método lança uma exceção HTTP com a mensagem de erro e o status HTTP 500 (Internal Server Error).

4.1.3.1.2 Services

Os Services contêm a lógica de negócios da aplicação, onde as regras e operações específicas são implementadas. O *BookService* é responsável por implementar essa lógica relacionada aos livros, incluindo a validação de dados e a interação com o repositório, contendo métodos essenciais para o gerenciamento de livros, como *create*, *findAll*, *findOne*, *update*, e *remove*. Estes métodos encapsulam a lógica necessária para realizar operações de CRUD (criar, ler, atualizar e excluir) de forma eficiente e segura.

A Figura 7 apresenta um exemplo do código do método *create* no *BookService*. Este método utiliza o *BookRepository* para criar um novo livro no banco de dados, onde ele recebe um objeto *createBookDto* com os dados do livro a ser criado e delega a operação de criação ao método *createBook* do repositório. Em caso de sucesso, o livro é criado e retornado; em caso de erro, uma exceção é lançada com uma mensagem de erro específica.

Figura 7 - Código do método create no *BookService*.

```
async create(createBookDto: CreateBookDto) {
  try {
    return await this.bookRepository.createBook(createBookDto);
  } catch (error) {
    throw new Error('Erro ao criar livro!');
  }
}
```

Fonte: Elaborada pela autora (2024).

4.1.3.1.3 Repositories

Lidam com a persistência de dados e a comunicação com o banco de dados. O Prisma ORM é utilizado para facilitar a interação com o banco de dados PostgreSQL, oferecendo uma abstração eficiente que permite a criação e manipulação de registros de forma simples e eficaz. A Figura 8 ilustra o exemplo do código do método POST no *Repository* de Livros, que é responsável por criar novos registros de livros.

Figura 8 - Código do método POST no *Repository* de Livros.

```
createBook(data: any) {
  return this.prisma.book.create({ data });
}
```

Fonte: Elaborada pela autora (2024).

No código exemplificado, o método *createBook* utiliza o Prisma ORM para criar um novo livro no banco de dados, recebendo um objeto data contendo os dados do livro a ser criado e utiliza o Prisma para realizar a operação de persistência, retornando o registro do livro recém-criado.

4.1.3.2 Estrutura do Projeto

A estrutura do diretório é projetada para promover uma separação clara entre diferentes camadas da aplicação, facilitando a escalabilidade e a manutenção, de acordo com a organização modular do framework NestJS. O diretório src/ serve como o diretório principal do código fonte da aplicação e organiza os componentes de maneira a refletir a arquitetura modular do projeto. Dentro do diretório principal, a estrutura de diretórios e arquivos é a seguinte:

- *controllers/*: Contém os *Controllers* responsáveis por gerenciar as requisições HTTP e interagir com os serviços, eles são a camada de entrada da aplicação, onde as rotas são definidas para atender operações CRUD. Exemplos:
    1. *book.controller.ts*: Lida com as operações de CRUD para os livros.
    2. *genre.controller.ts*: Lida com as operações de CRUD para os gêneros.
- *services/*: Contém os Services que implementam a lógica de negócios e fornecem funcionalidades específicas para os *Controllers*, onde interagem com os repositories para acessar ou manipular dados no banco de dados. Exemplos:

1. *book.service.ts*: Implementa a lógica de negócios para o módulo de livros.
2. *genre.service.ts*: Implementa a lógica de negócios para o módulo de gêneros.

- *repositories/*: Contém os *Repositories*, responsáveis pela comunicação com o banco de dados através do ORM Prisma. Esta camada isola a lógica de acesso ao banco de dados, permitindo que o serviço apenas faça chamadas para obter ou modificar os dados sem se preocupar com detalhes de implementação. Exemplos:
    1. *book.repository.ts*: Realiza operações CRUD no banco de dados para a entidade Book.
    2. *genre.repository.ts*: Realiza operações CRUD no banco de dados para a entidade Genre.
- *models/*: Contém as definições de entidades e DTOs (Data Transfer Objects) usados no projeto, que são usados para validar e tipar os dados de entrada e saída. Exemplos:
    1. *create-book.dto.ts*: Define a estrutura dos dados para criar um livro.
    2. *update-book.dto.ts*: Define a estrutura dos dados para atualizar um livro.
    3. *create-genre.dto.ts*: Define a estrutura dos dados para criar um gênero.
    4. *update-genre.dto.ts*: Define a estrutura dos dados para atualizar um gênero.
- *config/*: Contém arquivos de configuração, como a configuração de conexão com o banco de dados via Prisma, facilitando a separação da lógica de configuração da lógica de negócios. Exemplo:
    1. *prisma.service.ts*: Configuração de conexão com o banco de dados PostgreSQL usando Prisma ORM. Garante que o serviço Prisma esteja disponível para os serviços utilizarem.
- *app.module.ts*: O módulo principal da aplicação, responsável por importar os outros módulos, como *BookModule* e *GenreModule*. Esse arquivo faz a ligação entre os diferentes componentes do projeto, garantindo que os serviços, controllers e repositórios funcionem em conjunto.
- main.ts: O ponto de entrada da aplicação, responsável por inicializar e servir as rotas. Ele inicializa o servidor HTTP e expõe a API na porta definida.

4.1.4 API em ASP.NET Core

Nesta seção, é apresentada a implementação da API utilizando ASP.NET Core, destacando suas principais características e a estrutura modular adotada, semelhante ao Nest.JS, para manter uma padronização entre as APIs.

4.1.4.1 Arquitetura da API

A API desenvolvida em ASP.NET Core segue uma arquitetura modular, com uma separação clara entre os *Controllers*, *Services*, e *Repositories*, garantindo uma divisão eficiente das responsabilidades, o que facilita a manutenção, escalabilidade e organização do código. Os *Controllers* recebem as requisições HTTP, os *Services* processam a lógica de negócios, e os *Repositories* lidam com a persistência dos dados.

4.1.4.1.1 Controllers

Assim como no NestJS, os *Controllers* em ASP.NET Core são responsáveis por lidar com as requisições HTTP e enviar as respostas adequadas, atuando como a interface principal da API, interagindo com os serviços para aplicar a lógica de negócios e processar os dados corretamente antes de retorná-los ao cliente. Na Figura 9, é apresentado o exemplo do método POST no *BooksController*, que cuida da criação de novos registros de livros na base de dados.

Figura 9 - Método POST no *BooksController*.

```
[Route("[controller]")]
[ApiController]
1 reference
public class BooksController : ControllerBase
{
    2 references
    private readonly BookService _bookService;

    0 references
    public BooksController(BookService bookService)
    {
        _bookService = bookService;
    }

    [HttpPost]
    0 references
    public async Task<IActionResult> CreateBook([FromBody] Book book)
    {
        await _bookService.CreateBook(book);
        return CreatedAtAction(nameof(GetBook), new { id = book.Id }, book);
    }
}
```

Fonte: Elaborada pela autora (2024).

Nesse exemplo, o método *CreateBook* recebe os dados do livro através do corpo da requisição, que é representado pelo objeto Book. O serviço *BookService* é então chamado para processar esses dados e criar o novo livro no banco de dados. Após a criação, o método retorna o livro recém-criado junto com o status HTTP 201 (Created), usando o método *CreatedAtAction*, que também gera o link para o recurso recém-criado. Caso ocorra algum erro, o método trata a exceção e retorna um status de erro adequado.

4.1.4.1.2 Services

No ASP.NET Core, os Services encapsulam a lógica de negócios e atuam como intermediários entre os *Controllers* e os *Repositories*, contêm as regras de negócio e validam as operações antes de interagir com o banco de dados. Assim como no NestJS, os *Services* no ASP.NET Core são responsáveis por processar dados e garantir que as regras de negócio sejam seguidas antes de persistir ou recuperar informações do banco de dados. Na Figura 10, o exemplo do *BookService* mostra como o serviço lida com a atualização de um livro existente.

Figura 10 - Método *UpdateBook* no *BookService*.

```
public class BookService
{
    3 references
    private readonly IBookRepository _bookRepository;

    0 references
    public BookService(IBookRepository bookRepository)
    {
        _bookRepository = bookRepository;
    }
    1 reference
    public async Task UpdateBook(Book book)
    {
        var existingBook = await _bookRepository.GetBookById(book.Id);
        if (existingBook == null)
        {
            throw new Exception("Book not found");
        }

        await _bookRepository.UpdateBook(book);
    }

}
```

Fonte: Elaborada pela autora (2024).

No exemplo do código, o serviço verifica se o livro a ser atualizado existe no banco de dados e, em seguida, utiliza o *BookRepository* para realizar a operação. O método *UpdateBook* valida a existência do livro no banco de dados ao chamar o método *GetBookById* do repositório. Se o livro for encontrado, a atualização é realizada com o método *UpdateBook* do *BookRepository*, caso contrário, o serviço lança uma exceção para sinalizar que o livro não foi encontrado.

4.1.4.1.3 Repositories

Os *Repositories* são responsáveis por realizar as operações de persistência no banco de dados, fornecendo uma camada de abstração entre os serviços e o banco de dados, permitindo que as operações CRUD (Create, Read, Update, Delete) sejam executadas de forma isolada da lógica de negócios. Os repositórios garantem que o código responsável por acessar o banco de dados esteja separado do restante da aplicação, seguindo o princípio da separação de responsabilidades.

Na Figura 11, o *BookRepository* mostra um exemplo de como as operações de busca de livros são realizadas. O método *GetBookById* utiliza o *ApplicationDbContext* para buscar um livro com base no seu ID, retornando o livro encontrado ou lançando uma exceção caso o livro não seja encontrado.

Figura 11 - Método *GetBookById* no *BookRepository*.

```
public class BookRepository : IBookRepository
{
    2 references
    private readonly ApplicationDbContext _context;

    0 references
    public BookRepository(ApplicationDbContext context)
    {
        _context = context;
    }

    0 references
    public async Task<Book> GetBookById(int id)
    {
        var book = await _context.Books.FindAsync(id);

        return book ?? throw new KeyNotFoundException("Livro não encontrado.");
    }
}
```

Fonte: Elaborada pela autora (2024).

4.1.4.2 Estrutura do Projeto

A organização do projeto em ASP.NET Core segue uma abordagem modular, dividindo os componentes de acordo com suas responsabilidades, garantindo uma separação clara entre as diferentes camadas da aplicação. A seguir, está a estrutura de diretórios adotada no projeto:

- *Controllers/*: Contém os controladores que lidam com as requisições HTTP. Cada controlador é responsável por expor os endpoints da API e encaminhar as requisições para os serviços correspondentes. Exemplos:
    1. *BooksController.cs*: Controlador responsável pelas operações CRUD para a entidade Book.
    2. *GenresController.cs*: Controlador responsável pelas operações CRUD para a entidade Genre.
- *Data/*: É o diretório que contém o contexto do banco de dados, que é configurado no arquivo *ApplicationDbContext.cs*. Este arquivo herda de *DbContext* e configura as entidades do banco de dados, como Book e Genre.
- *Models/*: Contém as classes que representam as tabelas no banco de dados, definindo as propriedades das entidades. Exemplos:
    1. *Book.cs*: Modelo que representa a entidade Book, com propriedades como título, autor, quantidade de páginas, e a relação com o gênero do livro.
    2. *Genre.cs*: Modelo que representa a entidade Genre, com a propriedade nome.
- *Repositories/*: Contém as interfaces e implementações dos repositórios que abstraem as operações de banco de dados. Exemplos:
    1. *IBookRepository.cs*: Interface que define as operações de banco de dados para livros.
    2. *BookRepository.cs*: Implementação das operações de banco de dados para livros.
    3. *IGenreRepository.cs*: Interface que define as operações de banco de dados para gêneros.
    4. *GenreRepository.cs*: Implementação das operações de banco de dados para gêneros.

- *Services/*: Contém os serviços que encapsulam a lógica de negócio da aplicação, chamando os repositórios para realizar as operações no banco de dados. Exemplos:
    1. *BookService.cs*: Implementação das operações de negócio para livros.
    2. *GenreService.cs*: Implementação das operações de negócio para gêneros.
- *appsettings.json*: Este arquivo contém as configurações da aplicação, como strings de conexão do banco de dados, configurações de logging e outras opções de configuração que podem ser facilmente alteradas sem modificar o código.
- *Program.cs*: Este arquivo é o ponto de entrada da aplicação, onde a configuração do host e os serviços necessários para a execução da aplicação são definidos.

4.1.5 API em Go com Echo

Nesta seção, é apresentada a implementação da API utilizando Go com o framework Echo, destacando suas principais características e a estrutura modular adotada, semelhante ao Nest.JS e ao ASP.NET Core, para manter a padronização entre as APIs.

4.1.5.1 Arquitetura da API

A API desenvolvida em Go utilizando o framework Echo segue uma arquitetura modular bem definida, com a separação clara entre *handlers*, *services* e *repositories*. Essa abordagem facilita a manutenção e a escalabilidade, promovendo o princípio da separação de responsabilidades.

4.1.5.1.1 Handlers

Os *Handlers* são responsáveis por lidar com as requisições HTTP, eles recebem as solicitações dos clientes, chamam os serviços para aplicar a lógica de negócio, e retornam as respostas apropriadas. No contexto do Go com Echo, os *handlers* funcionam de maneira semelhante aos *controllers* em frameworks como ASP.NET Core e NestJS.

Na Figura 12, o método *GetBooks* do *BookHandler* processa uma requisição GET que busca todos os livros da base de dados. O método chama o serviço *BookService* para

recuperar a lista de livros. Se a operação for bem-sucedida, ele retorna a lista com o status HTTP 200 (Ok). Caso ocorra um erro, o método retorna uma resposta HTTP 500 (Internal Server Error) com a mensagem de erro.

Figura 12 - Método *GetBooks* no *BookHandler*.

```
func (h *BookHandler) GetBooks(c echo.Context) error {
    books, err := h.Service.GetBooks()
    if err != nil {
        return echo.NewHTTPError(http.StatusInternalServerError, err.Error())
    }
    return c.JSON(http.StatusOK, books)
}
```

Fonte: Elaborada pela autora (2024).

4.1.5.1.2 Services

Os *Services* contêm a lógica de negócios e atuam como intermediários entre os *Handlers* e os *Repositories*. Eles são responsáveis por aplicar as regras de negócio antes de interagir com o banco de dados, garantindo que os dados sejam manipulados de acordo com as políticas da aplicação.

Na Figura 13, o método *GetBooks* do *BookService* chama o método *FindAll* do *BookRepository* para recuperar a lista de livros do banco de dados. O serviço retorna a lista de livros junto com qualquer possível erro que possa ocorrer durante a operação.

Figura 13 - Método *GetBooks* no *BookService*.

```
func (s *BookService) GetBooks() ([]models.Book, error) {
    return s.Repo.FindAll()
}
```

Fonte: Elaborada pela autora (2024).

4.1.5.1.3 Repositories

Os *Repositories* no Go com Echo são responsáveis pela interação direta com o banco de dados. Eles abstraem as operações de persistência, permitindo que os *Services* e *Handlers* manipulem os dados sem se preocupar com os detalhes da execução das consultas.

Na Figura 14, o método *FindAll* do *BookRepository* usa o GORM para buscar todos os registros de livros no banco de dados. A função Preload("Genre") faz o pré-carregamento da

relação com o gênero de cada livro, garantindo que os dados completos estejam disponíveis na resposta. O método retorna a lista de livros e qualquer erro encontrado durante a operação.

Figura 14 - Método *FindAll* no *BookRepository*.

```
func (r *BookRepository) FindAll() ([]models.Book, error) {
    var books []models.Book
    err := r.DB.Preload("Genre").Find(&books).Error
    return books, err
}
```

Fonte: Elaborada pela autora (2024).

4.1.5.2 Estrutura do Projeto

A organização do projeto em Go segue uma abordagem modular, semelhante ao que é adotado em outros frameworks, mas com algumas convenções específicas do ecossistema Go. A separação clara entre *handlers*, *services*, *repositories* e *models* garante que cada componente tenha uma responsabilidade bem definida. Abaixo está a estrutura de diretórios utilizada no projeto:

- *config/*: Contém as configurações da aplicação, como a conexão com o banco de dados. Exemplo:
    1. *config.go*: Define a função *ConnectDB*, que utiliza o GORM para configurar a conexão com o banco de dados PostgreSQL e realizar as migrações automáticas das tabelas.
- *handlers/*: Contém os controladores responsáveis por lidar com as rotas HTTP e interagir com os serviços. São equivalentes aos controllers em outros frameworks. Exemplos:
    1. *book.handler.go*: Lida com as requisições relacionadas ao CRUD de livros.
    2. *genre.handler.go*: Lida com as requisições relacionadas ao CRUD de gêneros.
- *models/*: Contém a definição das estruturas de dados que serão mapeadas diretamente para as tabelas do banco de dados utilizando o GORM. Exemplos:
    1. *book.model.go*: Define a estrutura de dados do Book.
    2. *genre.model.go*: Define a estrutura de dados do Genre.

- *repositories/*: Contém as funções responsáveis por interagir diretamente com o banco de dados, realizando as operações de CRUD (Create, Read, Update, Delete). Exemplos:
    1. *book.repository.go*: Implementa as operações de CRUD para o modelo Book.
    2. *genre.repository.go*: Implementa as operações de CRUD para o modelo Genre.
- *services/*: Os serviços contêm as regras de negócio e orquestram as chamadas entre os repositórios e os handlers. Exemplos:
    1. *book.service.go*: Implementa a lógica de negócio para a entidade Book.
    2. *genre.service.go*: Implementa a lógica de negócio para a entidade Genre.
- *go.mod*: Arquivo que gerencia as dependências do projeto. Aqui são especificadas bibliotecas como Echo (para rotas HTTP) e GORM (para o ORM).
- *main.go*: Ponto de entrada da aplicação. Inicializa o servidor HTTP, conecta-se ao banco de dados, define as rotas e inicia a aplicação.

4.1.6 Repositório dos Códigos

Durante o desenvolvimento deste trabalho, as APIs RESTful implementadas nas três tecnologias escolhidas — Node.js, ASP.NET Core e Go — foram organizadas em um repositório no GitHub, contendo todos os arquivos de código-fonte, abrangendo as configurações de projeto, a estrutura dos diretórios e as implementações detalhadas de cada componente, como controladores, serviços e repositórios. Além disso, incluem-se todos os scripts de teste utilizados para cada API.

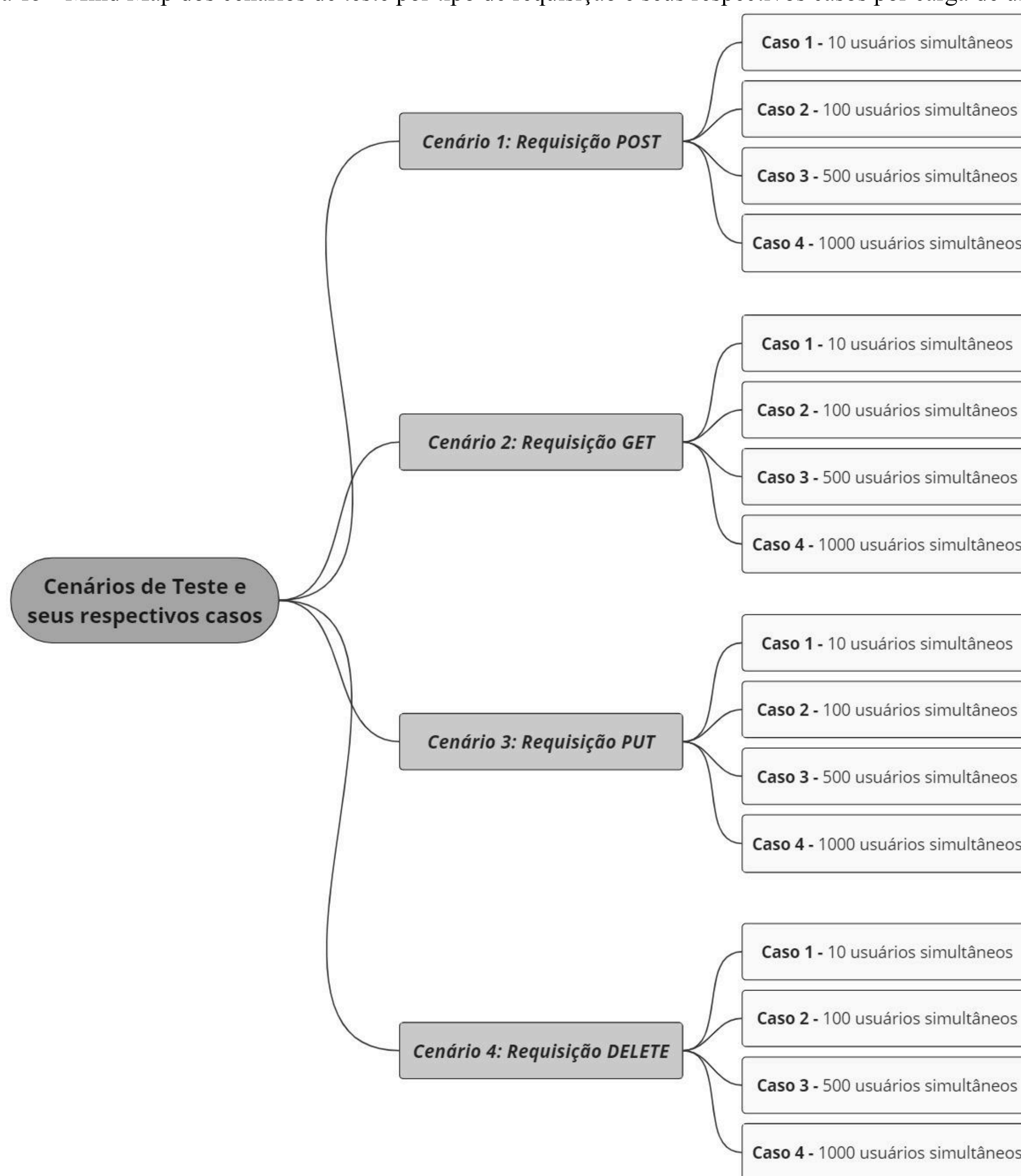
A disponibilização desse repositório tem como objetivo facilitar o acesso aos códigos utilizados nesta pesquisa, permitindo que outros desenvolvedores, estudantes e interessados possam visualizar e replicar a pesquisa, além de compreender as abordagens adotadas nas implementações. Para visualizar o código completo das APIs desenvolvidas, utilize o seguinte link:

https://github.com/mgGabrielly/tcc-apis.git

4.2 CENÁRIOS DE TESTE

Os cenários de teste e seus respectivos casos, exemplificados na Figura 15, foram elaborados para avaliar o desempenho das APIs desenvolvidas com diferentes tecnologias, utilizando a ferramenta k6 para medir o impacto de diversas cargas de usuários e possibilitar uma análise detalhada do desempenho em diferentes condições. Cada cenário foca em um tipo específico de requisição - GET, POST, PUT ou DELETE - e inclui casos com uma quantidade específica de usuários simultâneos, variando de 10 a 1000, permitindo uma comparação abrangente do impacto das diferentes tecnologias na eficiência e escalabilidade das APIs.

Figura 15 - Mind Map dos cenários de teste por tipo de requisição e seus respectivos casos por carga de usuários.



Fonte: Elaborada pela autora (2024).

Foram realizadas comparações extensivas entre as três APIs semelhantes, cada uma desenvolvida com uma tecnologia diferente: Node.js, ASP.NET Core e Go. Ao todo, foram realizados 48 testes, abrangendo todas as combinações de tipos de requisição e quantidades de usuários, permitindo uma ampla análise do impacto das diferentes tecnologias na eficiência e escalabilidade das APIs.

4.2.1 Cenário 1: Requisição POST

O objetivo do Cenário 1 é avaliar o desempenho das três APIs ao lidar com requisições de criação (POST) sob diferentes níveis de carga. Para cada API, foram realizados os quatro casos de testes distintos, cada um com uma carga específica de usuários simultâneos, e cada teste teve a duração de 3 minutos.

No total, foram realizados 12 testes neste cenário, com quatro testes para cada API. Espera-se que as respostas sejam retornadas com o status de sucesso 201 (Created).

4.2.2 Cenário 2: Requisição GET

O objetivo deste cenário é avaliar o desempenho das APIs ao lidar com requisições de leitura de dados (GET) sob diferentes níveis de carga de usuários. Os testes foram realizados de forma independente para cada nível de carga de usuários simultâneos, com duração de 3 minutos de cada teste.

A expectativa é de que as respostas das requisições retornem o status de sucesso 200 (Ok) para os quatros testes realizados para cada API, totalizando 12 testes neste cenário.

4.2.3 Cenário 3: Requisição PUT

O propósito deste cenário é avaliar a performance das APIs na atualização de dados existentes (PUT) sob diferentes níveis de carga de usuários simultâneos. Cada teste foi conduzido com uma carga específica de usuários, com a duração de 3 minutos por teste.

Espera-se que as respostas das requisições retornem o status 200 (Ok), o que confirmaria que a atualização dos dados ocorreu com sucesso e o recurso foi alterado. No total, foram realizados quatro testes para cada API, somando 12 testes neste cenário.

4.2.4 Cenário 4: Requisição DELETE

Este cenário tem como objetivo avaliar a performance das APIs ao executar operações de exclusão de dados (DELETE) sob diversas intensidades de carga de usuários simultâneos. Cada teste foi conduzido com um número específico de usuários e teve a duração de 3 minutos, permitindo a análise do comportamento das APIs em diferentes condições de carga.

A expectativa é que as respostas das requisições retornem o status 204 (No Content), indicando que a exclusão foi concluída com sucesso e sem necessidade de retornar conteúdo. Foram realizados quatro testes para cada API, totalizando 12 testes neste cenário.

4.3 DESENVOLVIMENTO DOS SCRIPTS DE TESTE

Para avaliar o desempenho das APIs, foi utilizado a ferramenta k6, desenvolvendo e configurando os scripts de teste, que têm o objetivo de simular diferentes tipos de requisições e medir o desempenho das APIs sob diversas condições de carga. A abordagem envolve a criação de scripts que definem a configuração dos testes, a lógica para realizar as requisições e a validação das respostas recebidas.

Os scripts de teste no k6 são elaborados em JavaScript, permitindo uma configuração flexível dos parâmetros necessários para simular as interações com a API. A estrutura básica dos scripts inclui a definição do número de usuários virtuais, a duração dos testes e os cenários de carga, com a lógica de cada script projetada para realizar requisições específicas e validar as respostas obtidas.

4.3.1 Script do Cenário 1: Requisição POST

O script desenvolvido para realizar requisições POST à API, especificamente para a criação de novos livros, foi projetado para verificar se o status de resposta indica sucesso. A Figura 16 ilustra o código completo deste script, fornecendo uma visão detalhada da configuração e da lógica utilizada. Todos os scripts de casos de teste seguem o mesmo formato básico, variando apenas o número de usuários simultâneos.

Figura 16 - Script do teste do caso 1, com 10 usuários no cenário 1, requisição POST.

```
import http from 'k6/http';
import { check, sleep } from 'k6';

export let options = {
  stages: [
    { duration: '3m', target: 10 },
  ],
};

export default function () {
  const url = 'http://localhost:3000/books';

  const uniqueTitle = `Livro-NPM-10-${__VU}-${__ITER}`;

  const payload = JSON.stringify({
    title: uniqueTitle,
    author: 'Autor',
    year: 2020,
    numberPages: 250,
    genreId: 'c86be521-d142-4a8e-9c27-1897309c69ca',
  });

  const params = {
    headers: {
      'Content-Type': 'application/json',
    },
  };

  const res = http.post(url, payload, params);

  const success = check(res, {
    'status é 201': (r) => r.status === 201,
  });

  sleep(1);
}
```

Fonte: Elaborada pela autora (2024).

Descrevendo cada parte do código, temos que:

- Importações: Importam os módulos necessários do k6 para realizar requisições HTTP (http), verificar resultados (check) e adicionar pausas (sleep).
- Configuração do Teste: Configura o teste para durar 3 minutos com 10 usuários virtuais simultâneos. O número de usuários virtuais foi ajustado conforme o caso de teste.
- URL da API: Define a URL para onde as requisições POST são enviadas.
- Dados da Requisição: A constante *uniqueTitle* gera um título único para cada livro usando variáveis do k6, enquanto que a constante payload cria o corpo da requisição com dados do livro em formato JSON.
- Cabeçalhos da Requisição: *params* define que o conteúdo da requisição é JSON.
- Envio da Requisição: Na constante *res* é enviada a requisição POST com o

corpo e cabeçalhos definidos.

- Verificação da Resposta: *success* verifica se a resposta da API tem o status 201 (Created), confirmando que o livro foi criado com sucesso.
- Intervalo entre Requisições: O *sleep*(1) adiciona uma pausa de 1 segundo entre as requisições para simular um intervalo realista.

4.3.2 Script do Cenário 2: Requisição GET

Para avaliar o desempenho da API em operações de leitura, foi desenvolvido um script para realizar requisições GET, focado na recuperação de dados de livros. A Figura 17 mostra o código detalhado deste script, oferecendo uma visão clara da configuração e da lógica empregada.

Figura 17 - Script do teste do caso 1, com 10 usuários no cenário 2, requisição GET.

```
import http from 'k6/http';
import { check, sleep } from 'k6';

export let options = {
  stages: [
    { duration: '3m', target: 10 },
  ],
};

export default function () {
  const url = 'http://localhost:3000/books';

  const res = http.get(url);

  const success = check(res, {
    'status é 200': (r) => r.status === 200,
  });

  sleep(1);
}
```

Fonte: Elaborada pela autora (2024).

Explicando cada parte do código, temos que:

- Importações: São importados os módulos necessários do k6 para realizar requisições HTTP (http), validar os resultados (check) e adicionar pausas (sleep).
- Configuração do Teste: Define que o teste deve durar 3 minutos com 10 usuários virtuais simultâneos, sendo ajustado conforme o caso de teste.
- URL da API: Especifica a URL de onde os dados dos livros serão recuperados

através das requisições GET.

- Envio da Requisição: *res* executa a requisição GET para a URL especificada.
- Verificação da Resposta: *success* verifica se a resposta tem o status 200 (Ok), garantindo que os dados foram recuperados com sucesso.
- Intervalo entre Requisições: Para simular um intervalo realista, o *sleep*(1) insere uma pausa de 1 segundo entre as requisições.

4.3.3 Script do Cenário 3: Requisição PUT

O script desenvolvido para realizar requisições PUT à API, focado na atualização de dados de livros, foi projetado para garantir que a API responda adequadamente e informe se a atualização foi bem sucedida. A descrição a seguir detalha cada parte do código antes de visualizar o código completo na Figura 18. O número de usuários simultâneos é o único dado que varia conforme o caso de teste específico.

- Importações: Importam os módulos necessários do k6 para realizar requisições HTTP (http), verificar resultados (check), agrupar requisições (group), e adicionar pausas (sleep).
- Função *getResourceIds*: Realiza uma requisição GET para obter IDs de recursos existentes na API, que são necessários para atualizar registros específicos, verificando se a resposta é bem sucedida e para extrair os IDs dos livros.
- Função *setup*: Chama *getResourceIds* para obter e retornar os IDs dos livros antes de iniciar os testes.
- Configuração do Teste: Define que o teste deve durar 3 minutos com 10 usuários virtuais simultâneos, ajustando conforme o caso de teste.
- URL da API: Especifica a URL para onde as requisições PUT serão enviadas para atualizar os dados dos livros, utilizando IDs extraídos anteriormente.
- Dados da Requisição: *payload* é criado com dados atualizados do livro em formato JSON, com um título único para cada requisição.
- Cabeçalhos da Requisição: params define que o conteúdo da requisição é JSON.
- Envio da Requisição: *putResponse* envia a requisição PUT com o corpo e cabeçalhos definidos para um ID específico.

- Verificação da Resposta: *success* verifica se a resposta da API possui o status 200 (Ok), confirmando que a atualização foi realizada com sucesso.
- Intervalo entre Requisições: *sleep*(1) adiciona uma pausa de 1 segundo entre as requisições para simular um intervalo realista.

Figura 18 - Script do teste do caso 1, com 10 usuários no cenário 3, requisição PUT.

```
import http from 'k6/http';
import { check, group, sleep } from 'k6';

let resourceIds = [];
let currentIndex = 0;

function getResourceIds() {
  const getResponse = http.get('http://localhost:3000/books');
  check(getResponse, { 'GET request is successful': (r) => r.status === 200, });
  const resources = getResponse.json();
  if (resources.length > 0) {
    return resources.map(resource => resource.id);
  } else {
    console.error('Nenhum recurso encontrado para atualização.');
    return [];
  }
}
export function setup() {
  resourceIds = getResourceIds();
  return { resourceIds };
}

export const options = { stages: [ { duration: '3m', target: 10 }, ], };

export default function (data) {
  const { resourceIds } = data;
  group('PUT book', function () {
    const idToUse = resourceIds[currentIndex];
    currentIndex = (currentIndex + 1) % resourceIds.length;
    const uniqueTitle = `Book-NPM-10-${__VU}-${__ITER}`;
    const payload = JSON.stringify({
      title: uniqueTitle,
      author: 'Author',
      year: 2021,
      numberPages: 280,
      genreId: 'c86be521-d142-4a8e-9c27-1897309c69ca',
    });
    const params = { headers: { 'Content-Type': 'application/json', }, };
    const putResponse = http.put(`http://localhost:3000/books/${idToUse}`, payload, params);
    check(putResponse, {
      'PUT request is successful': (r) => r.status === 200,
    });
    sleep(1);
  });
}
```

Fonte: Elaborada pela autora (2024).

4.3.4 Script do Cenário 4: Requisição DELETE

Para avaliar o desempenho da API em operações de exclusão, foi desenvolvido um script para realizar requisições DELETE, focado na remoção de livros. A Figura 19 mostra o código detalhado deste script, oferecendo uma visão clara da configuração e da lógica empregada. O código para DELETE é bastante semelhante ao do PUT, com a principal diferença sendo o tipo de operação HTTP executada.

Figura 19 - Script do teste do caso 1, com 10 usuários no cenário 4, requisição DELETE.

```
import http from 'k6/http';
import { check, group, sleep } from 'k6';

let resourceIds = [];
let currentIndex = 0;

function getResourceIds() {
  const getResponse = http.get('http://localhost:3000/books');
  check(getResponse, {
    'GET request is successful': (r) => r.status === 200,
  });
  const resources = getResponse.json();
  if (resources.length > 0) {
    return resources.map(resource => resource.id);
  } else {
    console.error('Nenhum recurso encontrado para exclusão.');
    return [];
  }
}
export function setup() {
  resourceIds = getResourceIds();
  return { resourceIds };
}
export const options = { stages: [ { duration: '3m', target: 10 }, ], };
export default function (data) {
  const { resourceIds } = data;
  group('DELETE book', function () {
    const idToUse = resourceIds[currentIndex];
    currentIndex = (currentIndex + 1) % resourceIds.length;
    const deleteResponse = http.del(`http://localhost:3000/books/${idToUse}`);
    check(deleteResponse, {
      'DELETE request is successful': (r) => r.status === 204,
    });
    sleep(1);
  });
}
```

Fonte: Elaborada pela autora (2024).

Descrevendo cada parte do código, temos que:

- Importações: Importam os módulos necessários do k6 para realizar requisições HTTP (http), verificar resultados (check), agrupar requisições (group), e adicionar pausas (sleep).
- Função *getResourceIds*: Realiza uma requisição GET para obter IDs de recursos existentes na API, que são necessários para excluir registros específicos.
- Função *setup*: Antes de iniciar os testes, é chamada *getResourceIds* para obter e retornar os IDs dos livros.
- Configuração do Teste: Define que o teste deve durar 3 minutos com 10 usuários virtuais simultâneos, ajustando conforme o caso de teste.
- URL da API: Especifica a URL para onde as requisições DELETE serão enviadas para remover os livros, utilizando IDs extraídos anteriormente.
- Envio da Requisição: *deleteResponse* envia a requisição DELETE com o ID do livro a ser excluído.
- Verificação da Resposta: *success* verifica se a resposta da API possui o status 204 (No Content), confirmando que a exclusão foi realizada com sucesso.
- Intervalo entre Requisições: *sleep*(1) adiciona uma pausa de 1 segundo entre as requisições para simular um intervalo realista.

4.4 EXECUÇÃO DOS TESTES

A execução dos testes é uma fase crucial para validar o desempenho das APIs com as diferentes tecnologias, com o processo abrangendo desde a preparação do ambiente até a coleta e análise dos resultados, foram realizadas as seguintes etapas:

1. Configuração do ambiente para assegurar o funcionamento correto de todas as APIs, incluindo a verificação de que cada instância estivesse acessível e operando conforme o esperado.
2. Os scripts de teste desenvolvidos para a ferramenta k6 também foram revisados e ajustados para garantir que estivessem prontos para simular as diferentes cargas de usuários.
3. Execução do comando de teste, conforme ilustrado na Figura 20. Onde é passado o caminho do arquivo de teste.

Figura 20 - Exemplo do comando de execução do teste com k6.

```
k6 run script-post-10.js
```

Fonte: Elaborada pela autora (2024).

Durante a execução dos testes, foram monitoradas métricas essenciais como o tempo de resposta e a taxa de sucesso das requisições. Após a conclusão de cada teste, os resultados foram exibidos no terminal, sendo coletados e registrados em arquivos para posterior análise. A Figura 21 mostra um exemplo dos resultados obtidos, conforme exibidos no terminal após a execução de um teste.

Figura 21 - Resultado exibido no terminal do teste no cenário 2 (GET) para o caso 1 (10 usuários).

```
checks.........................: 100.00% ✓ 653        ✗ 0
data_received..................: 21 MB   114 kB/s
data_sent......................: 56 kB   305 B/s
http_req_blocked...............: avg=27.61µs  min=0s       med=0s       max=5.34ms   p(90)=0s       p(95)=0s
http_req_connecting............: avg=8.76µs   min=0s       med=0s       max=1.19ms   p(90)=0s       p(95)=0s
http_req_duration..............: avg=387.63ms min=205.48ms med=391.54ms max=2.24s    p(90)=553.7ms  p(95)=608.68ms
  { expected_response:true }...: avg=387.63ms min=205.48ms med=391.54ms max=2.24s    p(90)=553.7ms  p(95)=608.68ms
http_req_failed................: 0.00%   ✓ 0          ✗ 653
http_req_receiving.............: avg=2.3ms    min=0s       med=1.53ms   max=12.89ms  p(90)=6.14ms   p(95)=7.01ms
http_req_sending...............: avg=30.37µs  min=0s       med=0s       max=640.1µs  p(90)=74µs     p(95)=218.81µs
http_req_tls_handshaking.......: avg=0s       min=0s       med=0s       max=0s       p(90)=0s       p(95)=0s
http_req_waiting...............: avg=385.3ms  min=205.28ms med=389.3ms  max=2.24s    p(90)=550.48ms p(95)=606.37ms
http_reqs......................: 653     3.587948/s
iteration_duration.............: avg=1.39s    min=1.21s    med=1.39s    max=3.26s    p(90)=1.55s    p(95)=1.61s
iterations.....................: 653     3.587948/s
vus............................: 1       min=1        max=9
vus_max........................: 10      min=10       max=10
```

Fonte: Elaborada pela autora (2024).

As principais métricas analisadas incluem o percentual de requisições que retornam com sucesso e as métricas apresentadas no *http_req_duration* relacionadas ao tempo de resposta, que são:

- Tempo Médio de Resposta (avg): Tempo médio que a API levou para responder a uma requisição.
- Tempo Mínimo de Resposta (min): O menor tempo registrado para uma resposta.
- Tempo Máximo de Resposta (max): O maior tempo registrado para uma resposta.

Essas métricas fornecem uma visão detalhada do desempenho das APIs sob diferentes

tecnologias, possibilitando uma análise comparativa da eficiência e escalabilidade das implementações. Ao avaliar o tempo de resposta, o percentual de sucesso das requisições e outras métricas de desempenho, é possível identificar qual tecnologia oferece o melhor desempenho para APIs RESTful.

Essa análise é fundamental para determinar o impacto das tecnologias na performance das APIs, ajudando a selecionar a solução mais adequada para atender aos requisitos de desempenho desejados, especialmente em relação ao tempo de resposta.

5. RESULTADOS

Neste capítulo, serão apresentados os resultados obtidos a partir dos testes de desempenho realizados nas APIs, incluindo a apresentação dos dados coletados, a discussão dos cenários testados, e a interpretação dos resultados observados. Cada cenário será descrito conforme as condições de carga aplicadas, correspondendo a cada caso dentro do cenário. Os resultados serão apresentados por meio de gráficos, seguidos por uma discussão crítica sobre o desempenho das APIs testadas utilizando a ferramenta k6.

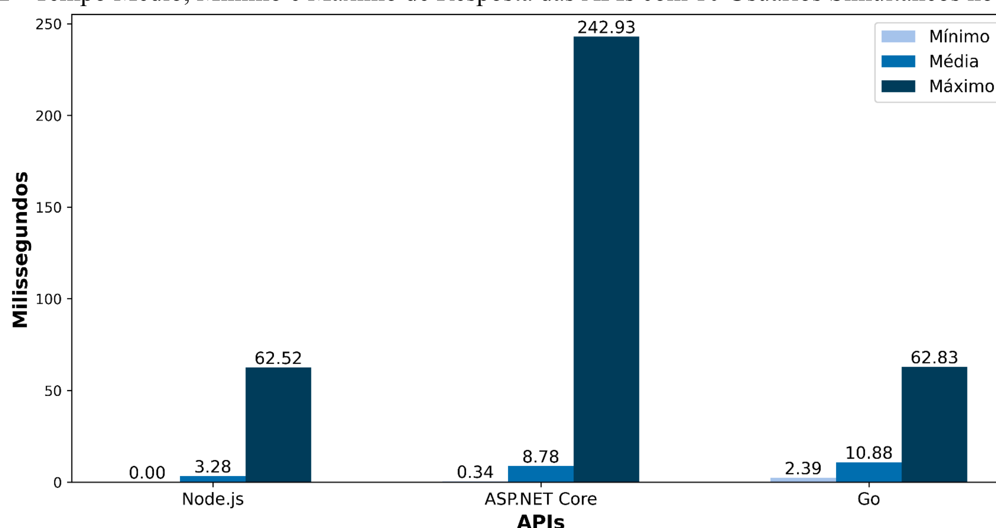
5.1 RESULTADOS DO CENÁRIO 1: MÉTODO POST

O Cenário 1 foi projetado para avaliar o desempenho das APIs ao lidar com requisições de criação de dados (POST) sob diferentes níveis de carga de usuários simultâneos. Para cada API, foram realizados quatro casos de teste com diferentes volumes de usuários, que serão apresentados nas próximas seções.

5.1.1 Resultados do Caso 1: 10 Usuários Simultâneos

A Figura 22 retrata o gráfico que apresenta o tempo médio de resposta das três APIs desenvolvidas em Node.js (NestJS), .NET (ASP.NET Core) e Go (Echo) ao processar requisições POST com 10 usuários simultâneos. A métrica observada foi o tempo de resposta médio, mínimo e máximo, medidos em milissegundos (ms), que indica a eficiência de cada API ao lidar com um volume leve de requisições simultâneas.

Figura 22 - Tempo Médio, Mínimo e Máximo de Resposta das APIs com 10 Usuários Simultâneos no Cenário 1.



Fonte: Elaborada pela autora (2024).

Comparando os resultados obtidos, observa-se que a API em Node.j teve o melhor desempenho médio com 3.28 ms, sendo mais rápida que as implementações em ASP.NET Core (8.78 ms) e Go (10.88 ms). Esse comportamento pode ser explicado pela capacidade do Node.js de processar requisições de forma assíncrona, o que é benéfico em cenários de baixa concorrência.

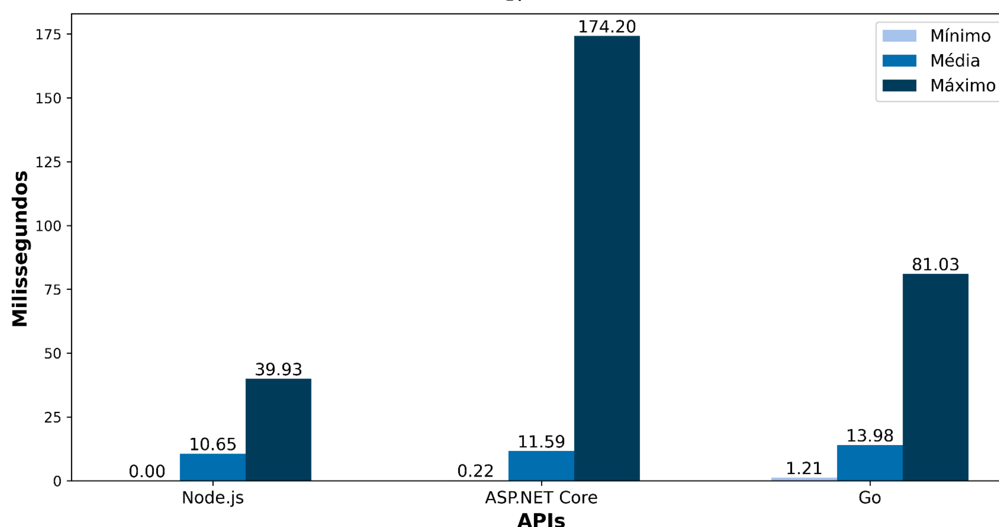
A API em ASP.NET Core apresentou um desempenho intermediário, o que foi superior ao Node.js. Este resultado pode ser atribuído à robustez do framework em lidar com requisições HTTP, especialmente em ambientes controlados.

Por outro lado, a API em Go registrou um desempenho mais lento entre as três implementações. No entanto, é importante destacar que o tempo máximo de resposta observado no Go foi menor, apresentando uma diferença mínima entre o tempo máximo (62.83 ms), médio (10.88 ms) e mínimo (2.39 ms). Essa consistência indica que, apesar de ser a implementação mais lenta em média, o Go demonstrou uma estabilidade significativa nos tempos de resposta, o que pode ser vantajoso em sistemas que requerem previsibilidade no desempenho.

5.1.2 Resultados do Caso 2: 100 Usuários Simultâneos

No Caso 2, as APIs foram submetidas a uma carga mais intensa, com 100 usuários simultâneos, para avaliar seu desempenho em um ambiente de concorrência maior. Na Figura 23 abaixo, apresenta os resultados mínimos, médios e máximos dos tempos de resposta para as APIs em Node.js, ASP.NET Core e Go.

Figura 23 - Tempo Médio, Mínimo e Máximo de Resposta das APIs com 100 Usuários Simultâneos no Cenário 1.



Fonte: Elaborada pela autora (2024).

No cenário de 100 usuários simultâneos, o Node.js manteve sua posição como a implementação mais rápida em termos de tempo médio (10.65 ms), mínimo (0 ms) e máximo (39.93 ms), sugerindo que continua a lidar bem com o aumento da carga, mantendo tempos mínimos muito baixos e tempos máximos dentro de uma faixa aceitável, apesar do aumento significativo de usuários.

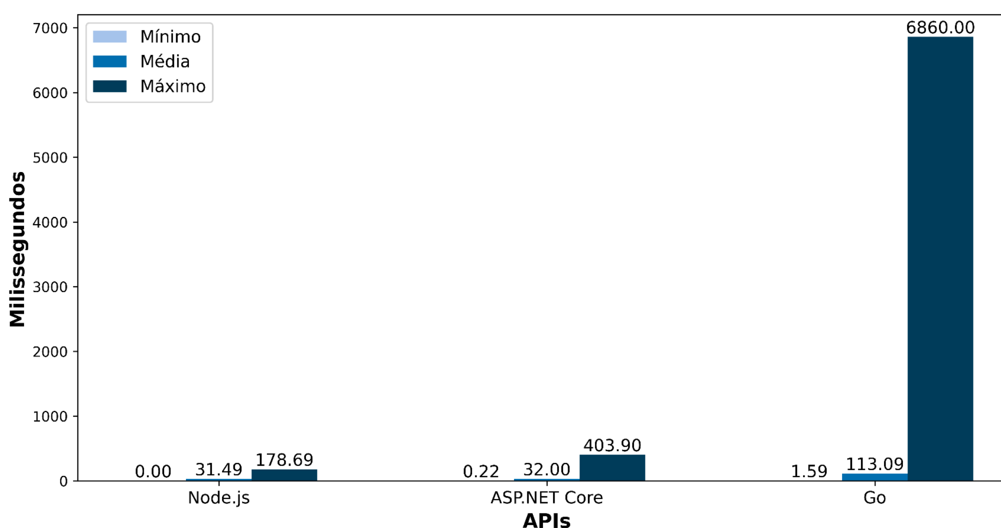
API em ASP.NET Core registrou um tempo médio de 11.59 ms, com um tempo mínimo de 0.2247 ms e um tempo máximo significativamente elevado de 174.2 ms. Embora o tempo médio seja competitivo com o do Node.js, o tempo máximo muito alto revela dificuldades em lidar com o aumento da carga, sugerindo que a API experimentou alguns picos de latência ao processar as requisições.

O maior tempo médio, de 13.98 ms, foi novamente da API em Go, que também apresentou um tempo mínimo de 1.21 ms e um tempo máximo de 81.03 ms. Apesar de ter o tempo médio mais alto entre as três implementações, o Go se destacou por sua estabilidade em relação ao ASP.NET Core, apresentando uma variação significativamente menor entre os tempos mínimo e máximo.

5.1.3 Resultados do Caso 3: 500 Usuários Simultâneos

Para o Caso 3, onde as APIs foram testadas com 500 usuários simultâneos, os resultados demonstram um aumento significativo nos tempos de resposta. A Figura 24 apresenta os dados mínimos, médios e máximos para cada implementação.

Figura 24 - Tempo Médio, Mínimo e Máximo de Resposta das APIs com 500 Usuários Simultâneos no Cenário 1.



Fonte: Elaborada pela autora (2024).

A API em Node.js apresentou um tempo médio de resposta de 31.49 ms, com um tempo mínimo de 0 ms e um máximo de 178.69 ms. Embora ainda tenha se destacado como a mais rápida, a latência aumentou em relação ao cenário anterior, evidenciando desafios em lidar com cargas mais pesadas.

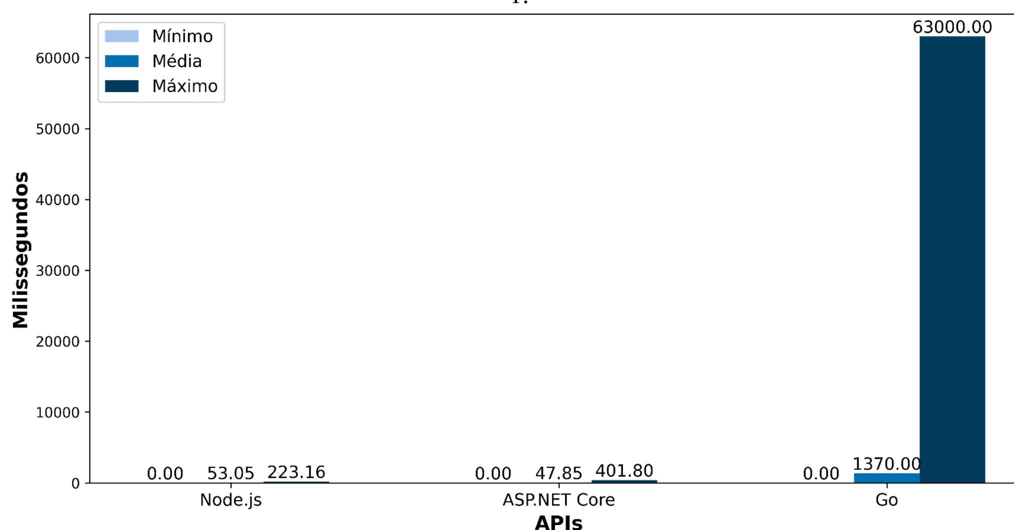
A API em ASP.NET Core teve um desempenho médio de 32 ms, com um mínimo de 0.2247 ms e um máximo elevado de 403.9 ms, indicando que, apesar do tempo médio competitivo, a latência máxima se tornou um problema significativo sob carga intensa.

Por fim, a API em Go apresentou o tempo médio mais alto, de 113.09 ms, com um mínimo de 1.59 ms e um máximo alarmante de 6860 ms, refletindo uma degradação acentuada no desempenho sob pressão, não conseguindo manter sua estabilidade em alta concorrência, levantando questões sobre sua adequação para sistemas que exigem desempenho previsível.

5.1.4 Resultados do Caso 4: 1000 Usuários Simultâneos

No Caso 4, com 1000 usuários simultâneos, os resultados indicam uma pressão ainda maior sobre as APIs, como mostrado na Figura 25.

Figura 25 - Tempo Médio, Mínimo e Máximo de Resposta das APIs com 1000 Usuários Simultâneos no Cenário 1.



Fonte: Elaborada pela autora (2024).

A API em Node.js registrou um tempo médio de resposta de 53.05 ms, com um mínimo de 0 ms e um máximo de 223.16 ms, mantendo-se relativamente rápida, mas enfrentando um aumento considerável nos tempos de resposta.

A API em ASP.NET Core apresentou um desempenho médio de 47.85 ms, com

mínimos de 0 ms e máximos de 401.8 ms, destacando-se como a mais rápida neste cenário, embora ainda enfrente desafios de latência máxima.

Por outro lado, a API em Go experimentou uma degradação severa, com um tempo médio de 1370 ms, mínimos de 0 ms e um máximo alarmante de 63000 ms. Essa disparidade extrema evidencia uma falha crítica em manter um desempenho aceitável sob alta concorrência, questionando a viabilidade do Go para aplicações que necessitam de respostas rápidas e consistentes.

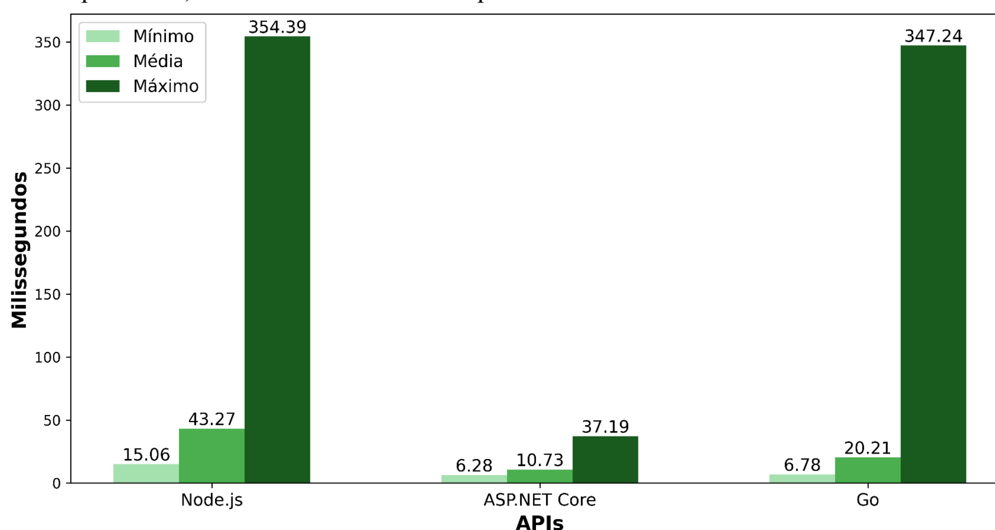
5.2 RESULTADOS DO CENÁRIO 2: MÉTODO GET

O Cenário 2 foi projetado para avaliar o desempenho das APIs ao lidar com requisições de leitura de dados (GET) sob diferentes níveis de carga de usuários simultâneos. Assim como no Cenário 1, foram realizados quatro casos de teste com volumes variados de usuários, que serão descritos nas seções seguintes.

5.2.1 Resultados do Caso 1: 10 Usuários Simultâneos

A Figura 26 apresenta o gráfico que ilustra o tempo médio de resposta das três APIs desenvolvidas em Node.js, ASP.NET Core e Go ao processar requisições GET com 10 usuários simultâneos. As métricas observadas incluem os tempos de resposta mínimo, médio e máximo, medidos em milissegundos (ms), que refletem a eficiência de cada API em um ambiente de baixa concorrência.

Figura 26 - Tempo Médio, Mínimo e Máximo de Resposta das APIs com 10 Usuários Simultâneos no Cenário 2.



Fonte: Elaborada pela autora (2024).

Os resultados revelam que a API em Node.js teve um tempo médio de resposta de 43.27 ms, com um mínimo de 15.06 ms e um máximo de 354.39 ms. Embora tenha apresentado um desempenho razoável, a latência máxima indica a possibilidade de picos sob determinadas condições.

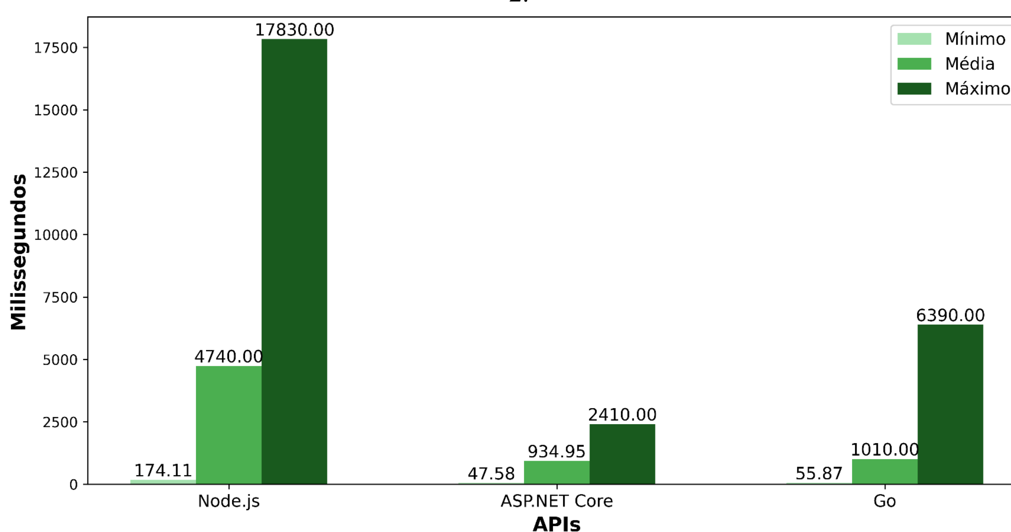
Em contraste, a API em ASP.NET Core destacou-se com um tempo médio de apenas 10.73 ms, um mínimo de 6.28 ms e um máximo de 37.19 ms, demonstrando uma eficiência superior sob essa carga leve.

Por fim, a API em Go apresentou um tempo médio de 20.21 ms, com mínimos de 6.78 ms e máximos de 347.24 ms, mostrando-se competitiva, mas não tão eficiente quanto o ASP.NET Core. Esses resultados indicam que, para cargas leves, o ASP.NET Core se sobressaiu em eficiência, enquanto o Node.js e o Go apresentaram variações maiores nos tempos máximos de resposta.

5.2.2 Resultados do Caso 2: 100 Usuários Simultâneos

A Figura 27 apresenta os dados de desempenho das APIs ao processar requisições GET com 100 usuários simultâneos. As métricas observadas incluem os tempos de resposta mínimo, médio e máximo, medidos em milissegundos (ms), refletindo a capacidade das APIs sob uma carga mais intensa.

Figura 27 - Tempo Médio, Mínimo e Máximo de Resposta das APIs com 100 Usuários Simultâneos no Cenário 2.



Fonte: Elaborada pela autora (2024).

Os resultados mostram que a API em Node.js teve um tempo médio de resposta alarmante de 4740 ms, com um mínimo de 174.11 ms e um máximo de 17830 ms,

evidenciando uma degradação significativa em relação ao cenário anterior.

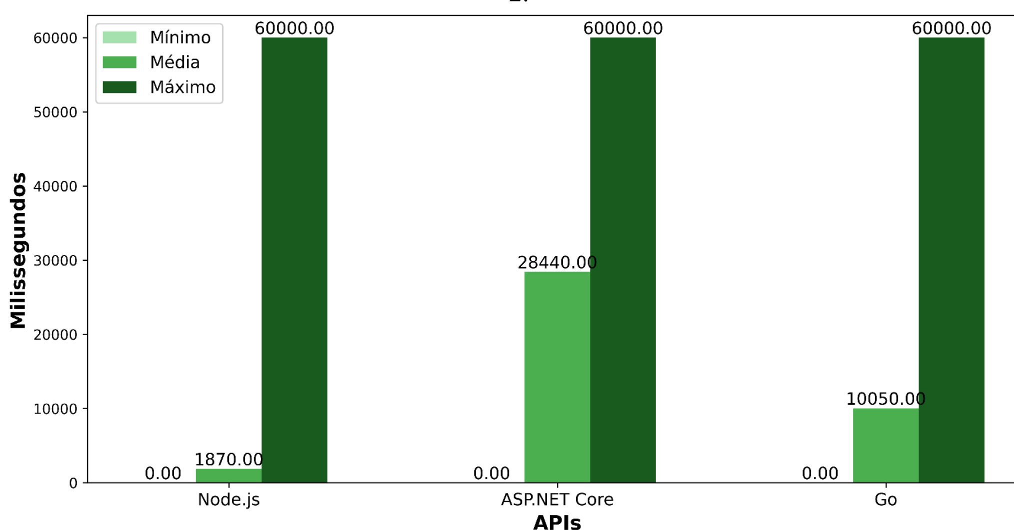
A API em ASP.NET Core apresentou um desempenho médio de 934.95 ms, com um mínimo de 47.58 ms e um máximo de 2410 ms, indicando uma capacidade melhor em lidar com a carga, mas ainda com tempos elevados.

A API em Go também apresentou tempos elevados, com um tempo médio de 1010 ms, mínimos de 55.87 ms e máximos de 6390 ms. O Node.js, em particular, mostrou a maior degradação de desempenho, sugerindo que pode não ser tão eficiente em cenários de alta concorrência.

5.2.3 Resultados do Caso 3: 500 Usuários Simultâneos

A Figura 28 apresenta os dados de desempenho das APIs ao processar requisições GET com 500 usuários simultâneos. As métricas observadas incluem os tempos de resposta mínimo, médio e máximo, medidos em milissegundos (ms), refletindo a capacidade das APIs sob uma carga ainda mais intensa.

Figura 28 - Tempo Médio, Mínimo e Máximo de Resposta das APIs com 500 Usuários Simultâneos no Cenário 2.



Fonte: Elaborada pela autora (2024).

Comparando os resultados, observa-se que a API em Node.js teve um desempenho médio de 1870 ms, com um mínimo de 0 ms e um máximo alarmante de 60000 ms. Esse aumento significativo nos tempos de resposta indica que o Node.js enfrentou sérios desafios sob alta concorrência, dificultando sua capacidade de escalar eficientemente.

A API em ASP.NET Core, por sua vez, apresentou um desempenho médio de 28440 ms, com mínimos de 0 ms e máximos também de 60000 ms. Apesar de ser a implementação

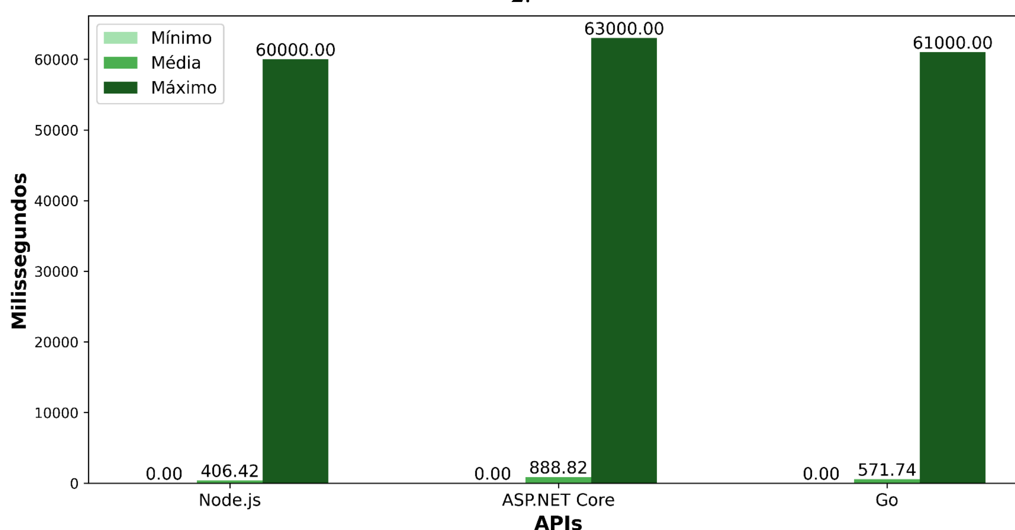
com maior tempo médio, sua capacidade de lidar com requisições em cenários de alta carga ficou clara, mas com uma latência preocupante.

Já a API em Go, com um tempo médio de 10050 ms, mínimos de 0 ms e máximos de 60000 ms, demonstrou uma degradação considerável, mas ainda inferior à do ASP.NET Core. Essa situação evidencia que, sob uma carga de 500 usuários, todas as implementações lutaram para manter um desempenho aceitável, ressaltando a necessidade de otimização em ambientes de alta concorrência.

5.2.4 Resultados do Caso 4: 1000 Usuários Simultâneos

Para o Caso 4, a análise do desempenho das APIs com 1000 usuários simultâneos é apresentada na Figura 29, onde é possível observar as métricas de tempos de resposta, incluindo os valores mínimos, médios e máximos, todos medidos em milissegundos (ms), que ilustram como cada API se comportou sob uma carga extremamente elevada.

Figura 29 - Tempo Médio, Mínimo e Máximo de Resposta das APIs com 1000 Usuários Simultâneos no Cenário 2.



Fonte: Elaborada pela autora (2024).

Os resultados mostram que a API em Node.js teve um tempo médio de resposta de 406.42 ms, com um mínimo de 0 ms e um máximo de 60000 ms. Apesar de apresentar uma melhoria em relação ao cenário anterior, a latência máxima ainda levanta preocupações sobre a capacidade de gerenciamento de picos de carga.

A API em ASP.NET Core, com um tempo médio de 888.82 ms, mínimos de 0 ms e máximos de 63000 ms, demonstrou um desempenho que, embora inferior ao Node.js, ainda apresenta desafios significativos em situações de alta concorrência. A latência máxima sugere

que houve dificuldades em manter a eficiência sob carga pesada.

A API em Go, por sua vez, registrou um tempo médio de 571.74 ms, com mínimos de 0 ms e máximos de 61000 ms. Seu desempenho médio foi melhor do que o do ASP.NET Core, mas a variação nos tempos máximos continua sendo uma preocupação. No geral, esses resultados ressaltam a necessidade de otimização para todas as implementações, especialmente em cenários de alta concorrência.

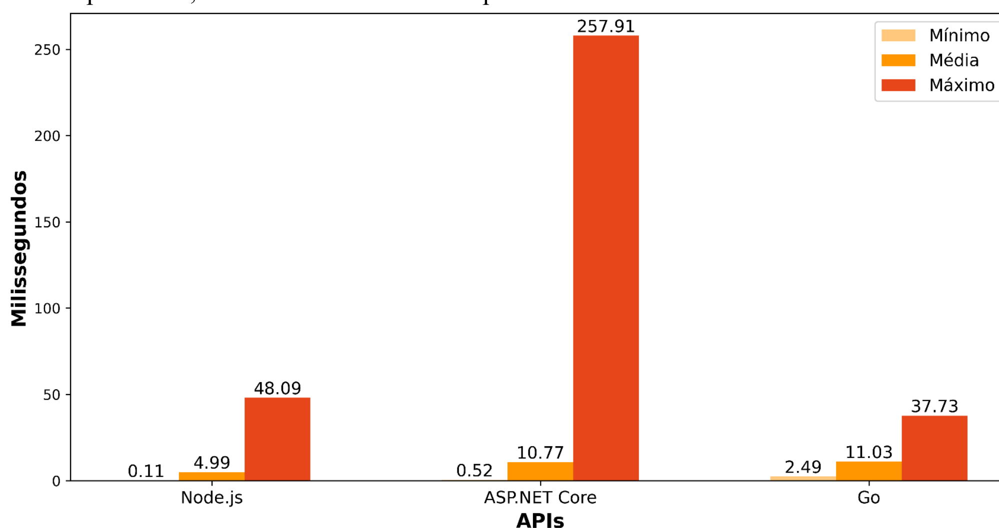
5.3 RESULTADOS DO CENÁRIO 3: MÉTODO PUT

O Cenário 3 foi elaborado para avaliar o desempenho das APIs ao processar requisições de atualização de dados (PUT) sob diferentes níveis de carga de usuários simultâneos. Assim como nos cenários anteriores, foram realizados quatro casos de teste com volumes variados de usuários, cujos resultados serão detalhados nas seções a seguir.

5.3.1 Resultados do Caso 1: 10 Usuários Simultâneos

A Figura 30 ilustra os dados de desempenho das APIs ao processar requisições PUT com 10 usuários simultâneos. As métricas observadas incluem os tempos de resposta mínimo, médio e máximo, medidos em milissegundos (ms), que refletem a eficiência das APIs em um ambiente de baixa concorrência.

Figura 30 - Tempo Médio, Mínimo e Máximo de Resposta das APIs com 10 Usuários Simultâneos no Cenário 3



Fonte: Elaborada pela autora (2024).

Os resultados revelam que a API em Node.js teve um tempo médio de resposta de 4.99 ms, com um mínimo de 0.1131 ms e um máximo de 48.09 ms. Esse desempenho sugere que a

API foi capaz de lidar eficientemente com a carga leve, mantendo tempos de resposta muito competitivos.

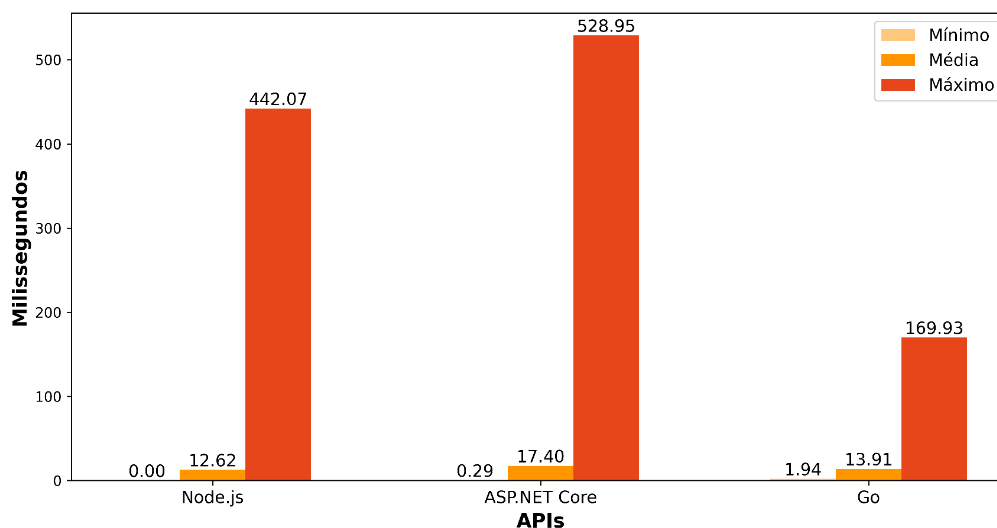
A API em ASP.NET Core apresentou um tempo médio de 10.77 ms, com um mínimo de 0.5165 ms e um máximo de 257.91 ms, mostrando-se eficaz, embora com uma variação maior no tempo máximo.

Por fim, a API em Go registrou um tempo médio de 11.03 ms, com mínimos de 2.49 ms e máximos de 37.73 ms, indicando que, apesar de um desempenho ligeiramente inferior à média, a implementação em Go se destacou pela consistência nos tempos de resposta.

5.3.2 Resultados do Caso 2: 100 Usuários Simultâneos

Neste segundo caso, foi analisado o desempenho das APIs ao processar requisições PUT com 100 usuários simultâneos. Os dados são apresentados na Figura 31 e revelam as métricas de tempo de resposta mínimo, médio e máximo em milissegundos (ms), proporcionando uma visão clara do desempenho sob carga mais intensa.

Figura 31 - Tempo Médio, Mínimo e Máximo de Resposta das APIs com 100 Usuários Simultâneos no Cenário 3.



Fonte: Elaborada pela autora (2024).

Os resultados mostram que a API em Node.js teve um tempo médio de resposta de 12.62 ms, com um mínimo de 0 ms e um máximo de 442.07 ms. Embora tenha se comportado bem, a latência máxima sugere que a API pode ter enfrentado desafios em alguns picos de carga.

A API em ASP.NET Core, com um tempo médio de 17.4 ms, mínimos de 0.2926 ms e máximos de 528.95 ms, demonstrou uma resposta relativamente sólida, mas a variação nos

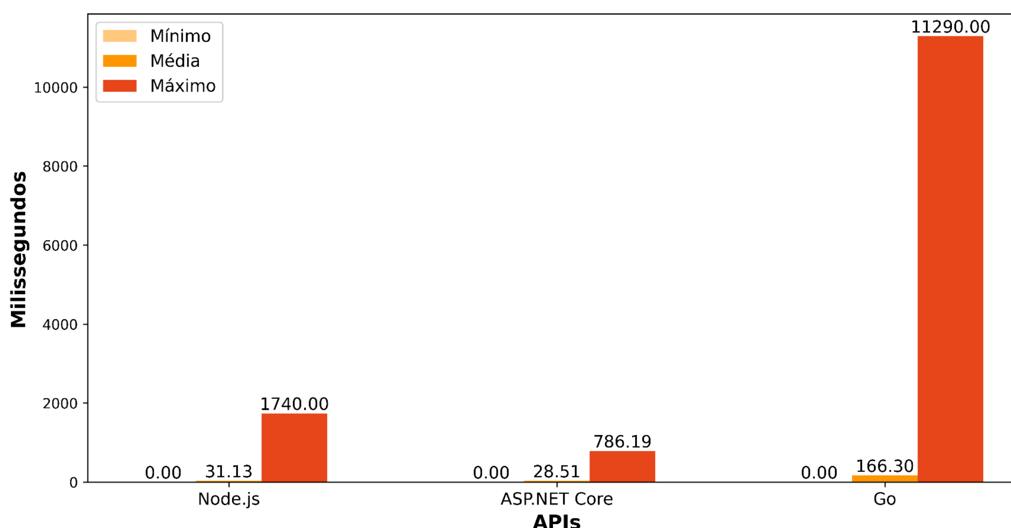
tempos máximos indica que pode ter enfrentado dificuldades em momentos de alta concorrência.

A API em Go, por sua vez, registrou um tempo médio de 13.91 ms, com mínimos de 1.94 ms e máximos de 169.93 ms. Embora tenha se destacado com um desempenho médio competitivo, a consistência nos tempos de resposta foi um ponto positivo, mostrando que a implementação é capaz de lidar com a carga de maneira eficiente.

5.3.3 Resultados do Caso 3: 500 Usuários Simultâneos

No Caso 3, o foco foi no desempenho das APIs ao lidar com requisições PUT em um cenário de 500 usuários simultâneos. Os resultados, ilustrados na Figura 32, apresentam os tempos de resposta mínimo, médio e máximo, medidos em milissegundos (ms), refletindo a capacidade das APIs de gerenciar essa carga significativa.

Figura 32 - Tempo Médio, Mínimo e Máximo de Resposta das APIs com 500 Usuários Simultâneos no Cenário 3.



Fonte: Elaborada pela autora (2024).

A API em Node.js se destacou com um tempo médio de 31.13 ms, apresentando um mínimo de 0 ms e um máximo de 1740 ms. Apesar de sua média competitiva, o tempo máximo elevado indica que a API pode ter enfrentado alguns desafios em momentos de pico de carga.

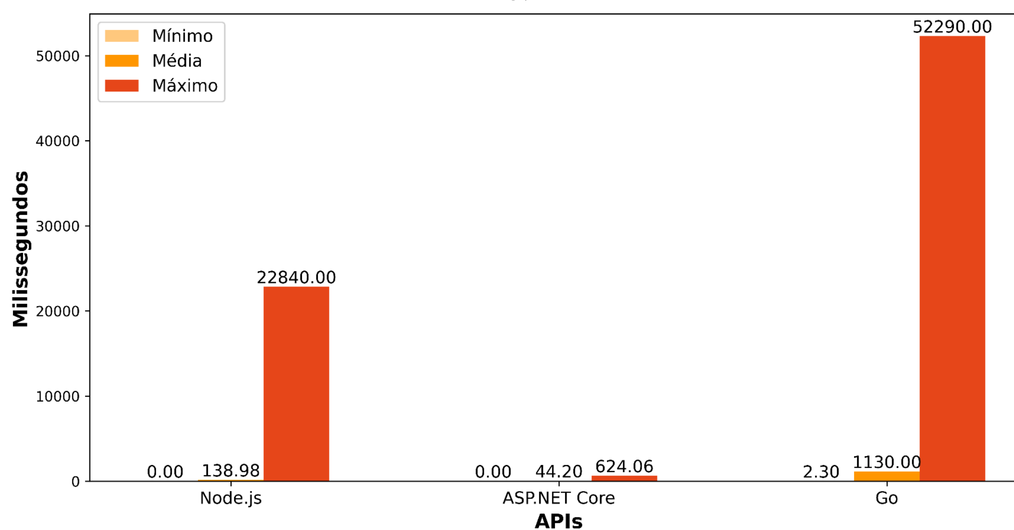
Por outro lado, a API em ASP.NET Core teve um desempenho médio de 28.51 ms, com mínimos de 0 ms e máximos de 786.19 ms. Esse resultado mostra que, embora tenha mantido um desempenho semelhante ao Node.js, a latência máxima sugere que também enfrentou dificuldades sob alta concorrência.

Já a API em Go, com um tempo médio de 166.3 ms, mínimos de 0 ms e máximos alarmantes de 11290 ms, apresentou um cenário preocupante, pois a grande variação nos tempos de resposta destaca uma degradação significativa sob carga intensa, levantando questões sobre sua adequação para cenários que exigem um desempenho estável.

5.3.4 Resultados do Caso 4: 1000 Usuários Simultâneos

Para o Caso 4, foi observado o desempenho das APIs ao processar requisições PUT com 1000 usuários simultâneos. Os resultados, apresentados na Figura 33, destacam os tempos de resposta mínimo, médio e máximo em milissegundos (ms), revelando como cada API se comportou sob uma pressão extrema.

Figura 33 - Tempo Médio, Mínimo e Máximo de Resposta das APIs com 1000 Usuários Simultâneos no Cenário 3.



Fonte: Elaborada pela autora (2024).

A API em Node.js teve um tempo médio de 138.98 ms, com um mínimo de 0 ms e um máximo impressionante de 22840 ms. Embora tenha mantido um desempenho relativamente bom, a latência máxima sugere que enfrentou dificuldades em momentos críticos.

A API em ASP.NET Core registrou um tempo médio de 44.2 ms, com mínimos de 0 ms e máximos de 624.06 ms. Esse desempenho mostra que, apesar de ser a mais rápida em comparação com as demais, também lidou com variações significativas durante a carga intensa.

Por fim, a API em Go apresentou um tempo médio de 1130 ms, com mínimos de 2.3 ms e um máximo alarmante de 52290 ms, indicando uma degradação severa, evidenciando a incapacidade de manter um desempenho aceitável sob alta concorrência.

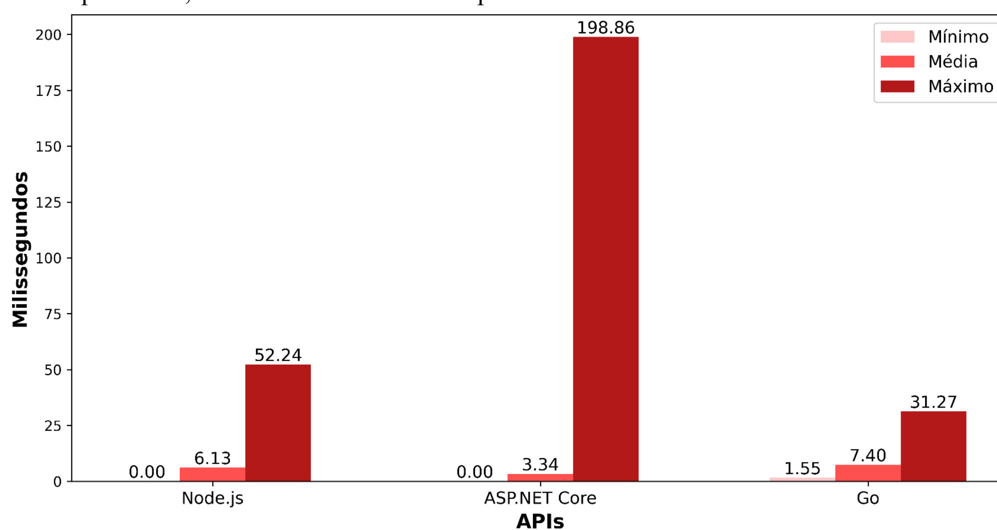
5.4 RESULTADOS DO CENÁRIO 4: MÉTODO DELETE

O quarto cenário foi elaborado para avaliar o desempenho das APIs ao processar requisições DELETE, considerando diferentes níveis de carga de usuários simultâneos. Para cada API, foram realizados testes em quatro casos distintos, começando com 10 usuários simultâneos.

5.4.1 Resultados do Caso 1: 10 Usuários Simultâneos

Os resultados, apresentados na Figura 34, mostram os tempos de resposta mínimo, médio e máximo em milissegundos (ms) para cada uma das APIs.

Figura 34 - Tempo Médio, Mínimo e Máximo de Resposta das APIs com 10 Usuários Simultâneos no Cenário 4.



Fonte: Elaborada pela autora (2024).

A API em Node.js teve um tempo médio de resposta de 6.13 ms, com um mínimo de 0 ms e um máximo de 52.24 ms, indicando que a API foi eficiente sob uma carga leve.

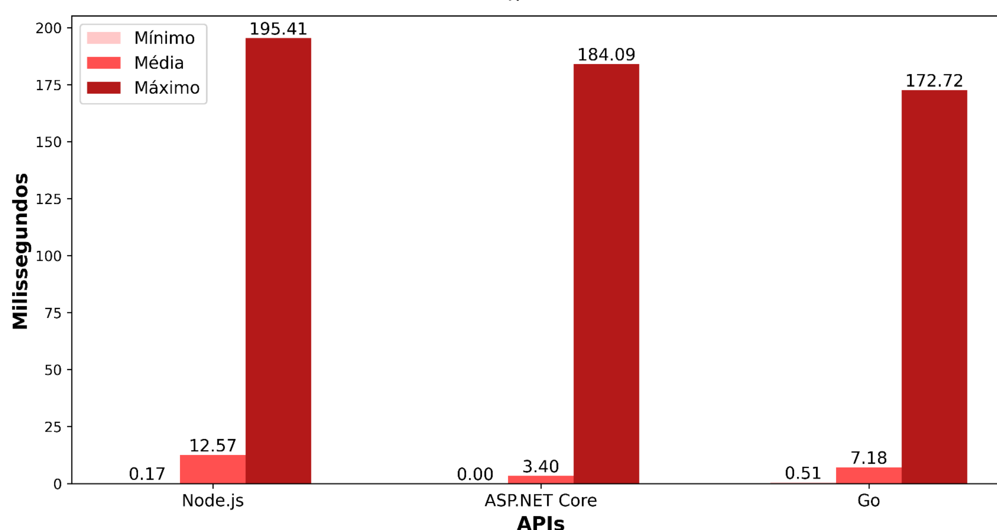
Porém, a API em ASP.NET Core destacou-se com um tempo médio de 3.34 ms, com mínimos de 0 ms e máximos de 198.86 ms, tornando-se a mais rápida do grupo, refletindo sua robustez em situações de baixa concorrência.

Por sua vez, a API em Go registrou um tempo médio de 7.4 ms, com um mínimo de 1.55 ms e um máximo de 31.27 ms. Apesar de ficar atrás das outras em média, ainda apresentou um desempenho aceitável para esse nível de usuários simultâneos.

5.4.2 Resultados do Caso 2: 100 Usuários Simultâneos

No segundo caso, foi analisado o desempenho das APIs ao lidar com requisições DELETE em um cenário de 100 usuários simultâneos. Os resultados, exibidos na Figura 35, oferecem uma visão abrangente dos tempos de resposta mínimo, médio e máximo em milissegundos (ms), permitindo uma comparação mais aprofundada entre as implementações.

Figura 35 - Tempo Médio, Mínimo e Máximo de Resposta das APIs com 100 Usuários Simultâneos no Cenário 4.



Fonte: Elaborada pela autora (2024).

Para a API em Node.js, o tempo médio de resposta foi de 12.57 ms, com um mínimo de 0.1732 ms e um máximo de 195.41 ms. Embora a média tenha apresentado um aumento em relação ao cenário anterior, o tempo mínimo ainda indica uma boa capacidade de resposta em condições leves de carga.

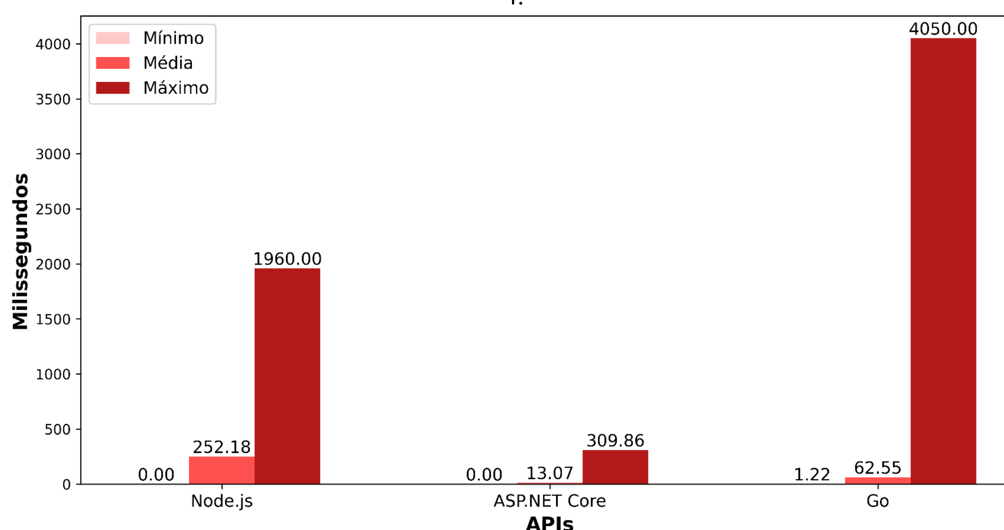
A API em ASP.NET Core teve um desempenho impressionante, com um tempo médio de apenas 3.4 ms. Com um mínimo de 0 ms e um máximo de 184.09 ms, essa implementação se destacou como a mais eficiente, mantendo uma latência baixa mesmo sob uma carga maior de usuários.

Por fim, a API em Go, por sua vez, apresentou um tempo médio de 7.18 ms, com um mínimo de 0.5148 ms e um máximo de 172.72 ms. Embora tenha melhorado em comparação ao cenário anterior, ainda não alcançou a eficiência das outras duas APIs, mas seu desempenho é aceitável considerando o aumento na concorrência.

5.4.3 Resultados do Caso 3: 500 Usuários Simultâneos

No terceiro caso, foi investigado o desempenho das APIs ao processar requisições DELETE com 500 usuários simultâneos. Os dados, apresentados na Figura 36, oferecem uma análise detalhada dos tempos de resposta mínimo, médio e máximo em milissegundos (ms), evidenciando como as diferentes implementações lidaram com uma carga substancial.

Figura 36 - Tempo Médio, Mínimo e Máximo de Resposta das APIs com 500 Usuários Simultâneos no Cenário 4.



Fonte: Elaborada pela autora (2024).

A API em Node.js mostrou um tempo médio de resposta de 252.18 ms, com um mínimo de 0 ms e um máximo de 1960 ms, mostrando um aumento na latência média, em comparação aos cenários anteriores, o que reflete a dificuldade em gerenciar requisições sob uma carga mais intensa. No entanto, o tempo mínimo de 0 ms indica que, em algumas situações, a API foi capaz de processar requisições rapidamente, mostrando sua capacidade de escalar.

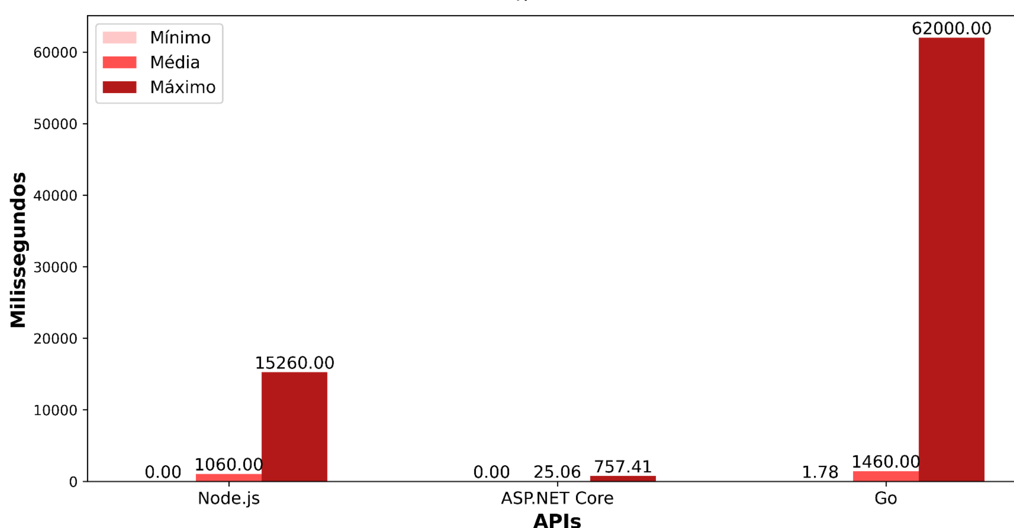
Por outro lado, a API em ASP.NET Core, apresentou um desempenho notável, com um tempo médio de apenas 13.07 ms. Com um mínimo de 0 ms e um máximo de 309.86 ms, essa implementação manteve uma excelente eficiência sob pressão, destacando-se pela robustez do framework em gerenciar concorrência.

A API em Go, embora tenha mostrado uma média de 62.55 ms, apresentou um mínimo de 1.22 ms e um máximo de 4050 ms. O aumento significativo no tempo máximo sugere uma degradação considerável sob alta carga, levantando preocupações sobre a consistência do desempenho em situações de estresse, o que pode impactar em aplicações que requerem respostas previsíveis e rápidas.

5.4.4 Resultados do Caso 4: 1000 Usuários Simultâneos

No quarto e último caso, foi avaliado o desempenho das APIs ao processar requisições DELETE sob uma carga extrema de 1000 usuários simultâneos. Os resultados, apresentados na Figura 37, oferecem uma análise abrangente dos tempos de resposta mínimo, médio e máximo em milissegundos (ms), refletindo como cada API se comportou sob pressão intensa.

Figura 37 - Tempo Médio, Mínimo e Máximo de Resposta das APIs com 1000 Usuários Simultâneos no Cenário 4.



Fonte: Elaborada pela autora (2024).

A API em Node.js registrou um tempo médio de resposta de 1060 ms, com um mínimo de 0 ms e um máximo de 15,260 ms. Embora ainda tenha demonstrado capacidade de resposta em algumas situações, o aumento significativo na latência média indica que a API teve dificuldades em lidar com a carga elevada, o que pode impactar a experiência do usuário em aplicações críticas.

Como destaque, a API em ASP.NET Core apresentou um tempo médio de apenas 25.06 ms, um mínimo de 0 ms e um máximo de 757.41 ms. Esse desempenho impressionante reforça a robustez do ASP.NET Core em cenários de alta concorrência, sugerindo que essa implementação é mais capaz de manter a eficiência mesmo quando submetida a pressões severas.

Por outro lado, a API em Go apresentou um tempo médio de 1460 ms, com um mínimo de 1.78 ms e um máximo alarmante de 62,000 ms. Essa disparidade significativa no tempo máximo evidencia uma falha crítica em manter um desempenho aceitável sob carga extrema, levantando sérias questões sobre a viabilidade da implementação em Go para aplicações que exigem respostas rápidas e consistentes.

5.5 ANÁLISE COMPARATIVA

Os resultados obtidos a partir dos testes de desempenho realizados nas APIs desenvolvidas em Node.js, ASP.NET Core e Go indicam diferenças consideráveis quanto à capacidade de cada tecnologia em lidar com diferentes níveis de carga de usuários simultâneos.

Nos cenários com baixa concorrência, como no teste com 10 usuários simultâneos, a API em Node.js apresentou os melhores tempos médios de resposta, beneficiando-se de sua arquitetura assíncrona, especialmente em requisições POST. Entretanto, conforme o número de usuários simultâneos aumentou, foi possível observar uma degradação acentuada no desempenho.

Em cenários com 500 e 1000 usuários simultâneos, a latência máxima disparou, sugerindo que o Node.js tem dificuldades em manter um desempenho estável sob alta concorrência. Portanto, para sistemas que exigem baixa latência em situações de carga leve, o Node.js é uma opção eficiente, mas para cenários de alta demanda, sua performance é comprometida.

A API desenvolvida em ASP.NET Core demonstrou um desempenho consistente, destacando-se principalmente em cenários de alta concorrência, mesmo sob cargas intensas, como nos testes com 500 e 1000 usuários simultâneos, a API manteve tempos médios de resposta competitivos, e embora tenha apresentado picos de latência máxima em alguns momentos, esses foram menores em comparação às outras tecnologias testadas. Essa robustez torna o ASP.NET Core uma escolha recomendada para sistemas que precisam lidar com grandes volumes de requisições simultâneas, sem comprometer a performance de forma significativa.

A API em Go apresentou um comportamento interessante ao longo dos testes, pois em cenários com baixa concorrência, o Go não foi tão eficiente quanto o Node.js ou ASP.NET Core, apresentando tempos médios de resposta mais altos. No entanto, a consistência nos tempos de resposta, com pouca variação entre os tempos mínimo e máximo, foi um ponto positivo nos testes com até 100 usuários simultâneos. Porém, à medida que a carga de usuários aumentou, a API em Go experimentou uma degradação significativa no desempenho, com picos de latência alarmantes no cenário com 1000 usuários, indicando que, para cenários de concorrência extrema, o Go necessita de otimizações adicionais para evitar comprometimentos severos na performance.

Com base nos testes realizados, a escolha da tecnologia mais adequada dependerá do

cenário de uso da aplicação. Se a prioridade for alta escalabilidade e estabilidade sob grandes volumes de requisições, o ASP.NET Core mostrou-se a melhor opção, oferecendo um equilíbrio entre desempenho e consistência. O Node.js é uma boa escolha para cenários de baixa carga, onde sua arquitetura assíncrona pode ser explorada ao máximo, mas não se mostrou tão eficiente sob alta concorrência. Por fim, o Go, embora ofereça previsibilidade nos tempos de resposta sob carga moderada, precisaria de ajustes significativos para ser viável em sistemas com alta demanda concorrencial.

6. CONCLUSÕES

Neste capítulo, serão apresentadas as considerações finais deste estudo, que analisou o desempenho de três tecnologias populares para o desenvolvimento de APIs RESTful: Node.js com NestJS, ASP.NET Core e Go com Echo. Além de resumir os principais resultados obtidos, serão discutidas as limitações da pesquisa e apresentadas sugestões para trabalhos futuros.

6.1 CONSIDERAÇÕES FINAIS

Este trabalho buscou avaliar o desempenho de três tecnologias populares no desenvolvimento de APIs RESTful: Node.js com NestJS, ASP.NET Core, e Go com Echo, utilizando a ferramenta k6 para simular diferentes cenários de carga e medir o tempo de resposta das APIs, proporcionando uma análise detalhada das suas capacidades e limitações. Os resultados revelaram variantes importantes sobre o comportamento de cada tecnologia sob diversas condições, oferecendo uma visão mais clara sobre suas aplicabilidades em diferentes contextos.

A principal contribuição desta pesquisa foi a comparação empírica e detalhada do desempenho das APIs em três tecnologias distintas. Ao realizar testes controlados com diferentes níveis de carga de usuários, este estudo forneceu *insights* importantes sobre o comportamento de cada plataforma, especialmente em termos de tempo de resposta e consistência sob alta concorrência.

Os resultados ressaltam a necessidade de considerar cuidadosamente a escolha da tecnologia em cenários de alta concorrência, especialmente para aplicações onde a latência é uma preocupação central. A performance da API em ASP.NET Core, em particular, a posiciona como uma opção sólida para sistemas que precisam suportar grandes volumes de requisições simultâneas.

Essa pesquisa também contribuiu para a literatura ao oferecer uma análise prática e replicável para desenvolvedores e arquitetos de software que precisam tomar decisões informadas sobre a escolha de tecnologias para o desenvolvimento de APIs escaláveis e eficientes. A utilização de ferramentas amplamente aceitas, como o k6, torna este estudo útil para a comunidade, pois fornece um modelo para futuras análises de desempenho.

6.2 LIMITAÇÕES DA PESQUISA

Como em qualquer estudo empírico, este trabalho apresentou algumas limitações. Em

primeiro lugar, os experimentos foram realizados em um ambiente controlado, com especificações de hardware fixas, o que pode não refletir totalmente os desafios enfrentados em ambientes de produção com diferentes arquiteturas e infraestruturas.

Outro ponto a ser destacado é que o foco foi exclusivamente no tempo de resposta como métrica de desempenho. Embora essa métrica seja fundamental, outros fatores, como uso de memória e escalabilidade horizontal, poderiam complementar a análise e fornecer uma visão mais completa das capacidades de cada tecnologia.

6.3 SUGESTÕES PARA TRABALHOS FUTUROS

Para futuros estudos, sugere-se ampliar o escopo da pesquisa em algumas direções:

1. Exploração de outras tecnologias: Investigar o desempenho de outras linguagens e frameworks emergentes para APIs RESTful, como Rust ou Python com FastAPI, para ampliar o escopo de comparação.
2. Incluir outras arquiteturas de APIs: Analisar o uso de arquiteturas como GraphQL e gRPC, permitiria uma análise mais abrangente, comparando não apenas tecnologias, mas também estilos de comunicação entre sistemas.
3. Impacto de outras métricas de desempenho: Avaliar o uso de CPU e consumo de memória, proporcionando uma análise mais detalhada do comportamento das tecnologias testadas.
4. Impacto de latência e rede: Realizar testes que simulam diferentes condições de rede e latência para analisar como essas variáveis afetam o desempenho das APIs.
5. Uso de containers com Docker: Investigar como cada tecnologia pode ser implantada em ambientes Docker, incluindo a criação de subredes para simular diferentes configurações de rede e avaliar o desempenho sob essas condições.

REFERÊNCIAS

ARAÚJO, Maria Gabrielly de Almeida. **tcc-codes.** 2024. Disponível em: https://github.com/mgGabrielly/tcc-codes.git. Acesso em: 01 out. 2024.

BITTENCOURT, André Luiz de Moura Ramos. **Uma comparação de performance entre arquitetura GraphQL e REST.** Brasília: Universidade de Brasília, 2021. 54 f. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) – Instituto de Ciências Exatas, Departamento de Ciência da Computação, Universidade de Brasília, 2021. Disponível em: https://bdm.unb.br/bitstream/10483/31169/1/2021_AndreLuizRamosBittencourt_tcc.pdf. Acesso em: 17 ago. 2024.

CAMPBELL, Donald T; STANLEY, Julian C. **Experimental and quasi-experimental designs for research.** [S.l.]: Ravenio Books, 2015. Disponível em: https://www.sfu.ca/~palys/Campbell&Stanley-1959-Exptl&QuasiExptlDesignsForResearch.pdf. Acesso em: 23 jun. 2024.

CARMO, Klayver Ximenes. **Um estudo comparativo entre tecnologias de back-end:** Node.js, Django REST Framework e ASP.NET Core. 2023. 106 f. Trabalho de conclusão de curso (Graduação em Engenharia de Computação) Universidade Federal do Ceará, Campus de Sobral, Sobral, 2023. Disponível em: http://repositorio.ufc.br/handle/riufc/77574. Acesso em: 14 ago. 2024.

CHIMUCO, Pedro Ventura Lucunde; BARBOSA, Ana Claudia Garcia. **Validação de carga em testes de desempenho de APIs de cadastro de usuários:** Garantindo qualidade e eficiência com Docker e K6. 2024. Trabalho de Conclusão de Curso (Graduação em Engenharia de Software) – Universidade do Extremo Sul Catarinense, Criciúma, 2024. Disponível em: http://repositorio.unesc.net/handle/1/10988. Acesso em: 26 ago. 2024.

CHINA, Chrystal R. **GraphQL vs. REST API:** What's the difference? IBM Blog, 29 mar. 2024. Disponível em: https://www.ibm.com/blog/graphql-vs-rest-api. Acesso em: 14 ago. 2024.

COZBY, P. **Métodos de Pesquisa em Ciências do Comportamento.** 5. ed. São Paulo: Editora Atlas S.A., 2012. Disponível em: https://edisciplinas.usp.br/pluginfile.php/7920584/mod_resource/content/3/Cozby_P._C._2003_._Metodos_de_pesquisa_e.pdf. Acesso em: 24 jun. 2024.

DALBARD, Axel; ISACSON, Jesper. **Comparative study on performance between ASP.NET and Node.js Express for web-based calculation tools.** 2021. Trabalho de Conclusão de Curso (Graduação em Engenharia da Computação) – Jönköping University, Jönköping, 2021. Disponível em: https://www.diva-portal.org/smash/record.jsf?pid=diva2%3A1572101&dswid=1215. Acesso em: 17 set. 2024.

DEMIR, Dennis; NILSSON, Edward. **Performance comparison of REST vs GraphQL in different web environments:** Node.js and Python. Västerås: Mälardalen University, 2023. 29 f. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) – School of Innovation, Design and Engineering, Mälardalen University, 2023. Disponível em: https://urn.kb.se/resolve?urn=urn:nbn:se:lnu:diva-121877. Acesso em: 22 ago. 2024.

DHALLA, H. K. A performance comparison of RESTful applications implemented in Spring Boot Java and MS.NET Core. **Journal of Physics:** Conference Series, v. 1933, p. 012041, 2021. Disponível em: https://doi.org/10.1088/1742-6596/1933/1/012041. Acesso em: 22 jun. 2024.

ECHO. **Echo Documentation.** [s.d.]. Disponível em: https://echo.labstack.com/. Acesso em: 17 set. 2024.

EHSAN, A.; ABUHALIQA, M. A. M. E.; CATAL, C.; MISHRA, D. RESTful API testing methodologies: rationale, challenges, and solution directions. **Applied Sciences**, v. 12, n. 9, p. 4369, 2022. DOI: 10.3390/app12094369. Disponível em: https://doi.org/10.3390/app12094369. Acesso em: 17 ago. 2024.

FIELDING, Roy et al. **Hypertext Transfer Protocol–HTTP/1.1.** 1999. Disponível em: https://www.rfc-editor.org/rfc/rfc2616?data1=dwnsb4B&data2=abmurltv2b. Acesso em: 18 ago. 2024.

FIELDING, Roy Thomas. **Architectural styles and the design of network-based software architectures.** 2000. Tese (Doutorado em Ciência da Computação) — University of California, Irvine, 2000. Disponível em: https://ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf. Acesso em: 10 ago. 2024.

GO. **GO FOR WEB DEVELOPMENT.** 2019. Disponível em: https://go.dev/solutions/webdev. Acesso em: 17 set. 2024.

GOLMOHAMMADI, A.; ZHANG, M.; ARCURI, A. Testing RESTful APIs: a survey. **ACM** Transactions on Software Engineering and Methodology, v. 33, n. 1, Art. 27, p. 1-41, nov. 2023. DOI: 10.1145/3617175.

GOODWIN, Michael. **O que é uma API (interface de programação de aplicativos)?** IBM, 09 abr. 2024. Disponível em: https://www.ibm.com/br-pt/topics/api. Acesso em: 12 ago. 2024.

GRAFANA. Grafana k6. [s.d.]. **Grafana Documentation.** Disponível em: https://grafana.com/docs/k6/latest/. Acesso em: 26 ago. 2024.

GRAY, D. E. **Pesquisa no mundo real.** 2. ed. Porto Alegre: Penso, 2012. Disponível em: https://www.academia.edu/24859386/Livro_Pesquisa_no_Mundo_Real_David_E_Gray. Acesso em: 23 jun. 2024.

JONSSON, Max; QVARNSTRÖM, Eric. **A performance comparison on REST-APIs in Express.js, Flask and ASP.NET Core.** Västerås, Sweden: Mälardalen University, 2022. 40 f. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) – School of Innovation, Design and Engineering, Mälardalen University, 2022. Disponível em: https://www.diva-portal.org/smash/get/diva2:1669487/FULLTEXT01.pdf. Acesso em: 17 ago. 2024.

KARLSSON, Oliver. **A performance comparison between ASP.NET Core and Express.js for creating Web APIs.** 2021. Trabalho de Conclusão de Curso (Graduação em Ciência da

Computação) – Jönköping University, Jönköping, 2021. Disponível em: https://www.diva-portal.org/smash/record.jsf?pid=diva2%3A1586295&dswid=-8074. Acesso em: 17 set. 2024.

KRYLOV, Georgiy; PATROU, Maria; DUECK, Gerhard W.; SIU, Joran. **The Evolution of Garbage Collection in V8:** Google's JavaScript Engine. *In*: 2020 9th Mediterranean Conference on Embedded Computing (MECO), 8-11 junho 2020, Budva, Montenegro. IEEE, 2020. p. 1-8. Disponível em: https://www.doi.org/10.1109/MECO49872.2020.9134326. Acesso em: 22 ago. 2024.

LEITE, Gabriel. **Saiba o que é JSON e como utilizar.** Alura, 05 set. 2023. Disponível em: https://www.alura.com.br/artigos/o-que-e-json. Acesso em: 18 ago. 2023.

LOUZADA, Vinícius; CARVALHO, Caroline; LARANJA, Emerson. **API:** o que é, para quê serve e qual é a sua importância. Alura, 01 mar. 2024. Disponível em: https://www.alura.com.br/artigos/api. Acesso em: 14 ago. 2024.

MAIOR, Milton José Vieira Souto. **Análise Comparativa de Performance de Frameworks para APIs Rest.** 2023. 55 f. Trabalho de Conclusão de Curso (Bacharelado em Sistemas de Informação) – Centro de Informática, Universidade Federal de Pernambuco, 2023. Disponível em:https://repositorio.ufpe.br/bitstream/123456789/50396/9/TCC_MiltonJoseVieiraSoutoMaior_mjvsm.pdf. Acesso em: 18 ago. 2024.

MARCONI, M. d. A.; LAKATOS, E. M. **Metodologia do trabalho científico.** 8ª. ed. São Paulo-SP: Editora Atlas, 2017. Disponível em: https://edisciplinas.usp.br/pluginfile.php/7237618/mod_resource/content/1/Marina%20Marconi%2C%20Eva%20Lakatos_Fundamentos%20de%20metodologia%20cient%C3%ADfica.pdf. Acesso em: 24 jun. 2024.

MASO, Nicolas Nascimento. **Comparativo entre arquiteturas de APIs - REST, GraphQL e gRPC.** 2024. Trabalho de Conclusão de Curso (Bacharelado em Sistemas de Informação) – Universidade Federal de Santa Catarina, Florianópolis, 2024. Disponível em: https://repositorio.ufsc.br/bitstream/handle/123456789/255699/TCC%20-%20Nicolas%20Nascimento%20Maso.pdf?sequence=1&isAllowed=y. Acesso em: 28 ago. 2024.

MIAZAKI, Alison. **GraphQL, REST, RPC e SOAP?** Medium, 19 mar. 2021. Disponível em: https://alisonmiazaki.medium.com/graphql-rest-rpc-e-soap-79a361e1a59e. Acesso em: 14 ago. 2024.

MICROSOFT. **Overview of ASP.NET Core.** 2024. Disponível em: https://learn.microsoft.com/en-us/aspnet/core/introduction-to-aspnet-core?view=aspnetcore-8.0. Acesso em: 17 set. 2024.

MORAES, Edmilson. **K6 x JMeter:** comparativo de ferramentas para testes de carga. Blog Vericode, 29 ago. 2023. Disponível em: https://blog.vericode.com.br/ferramentas-teste-de-carga-k6-jmeter/. Acesso em: 26 ago. 2024.

MOZILLA. **Uma visão geral do HTTP.** Disponível em: https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Overview. Acesso em: 18 ago. 2024.

NESTJS. **Nestjs Documentação.** [s.d.]. Disponível em: https://docs.nestjs.com/. Acesso em: 22 ago. 2024.

PEREIRA, Caio Ribeiro. **Aplicações web real-time com Node.js.** São Paulo: Casa do Código, 2014. 202 p.

PEROVANO, Dalton Gean. **Manual de metodologia da pesquisa científica.** 1. ed. Curitiba: Intersaberes, 2016. E-book. Disponível em: https://plataforma.bvirtual.com.br. Acesso em: 24 jun. 2024.

PETROSYAN, A. **Global number of internet users 2005-2023.** Statista, 22 maio 2024. Disponível em: https://www.statista.com/statistics/273018/number-of-internet-users-worldwide/. Acesso em: 18 set. 2024.

PROVDANOV, C. C.; FREITAS, E. C. de. **Metodologia do trabalho científico:** métodos e técnicas da pesquisa e do trabalho acadêmico. Novo Hamburgo: Feevale, 2013. Disponível em: https://doi.org/10.1017/CBO9781107415324.004. Acesso em: 24. jun. 2024.

RAZA, Syed Muhammad Ali. **API (Application Programming Interfaces) Types.** DEV Community, 10 set. 2023. Disponível em: https://dev.to/syedmuhammadaliraza/api-application-programming-interfaces-types-k3g. Acesso em: 14 ago. 2024.

SABO, Mario. **NestJS.** Osijek, Croácia: Josip Juraj Strossmayer University of Osijek, 2020. 42 f. Trabalho de Conclusão de Curso (Bacharelado em Matemática e Ciência da Computação) – Departamento de Matemática, Josip Juraj Strossmayer University of Osijek, 2020. Documento em croata. Disponível em: https://zir.nsk.hr/islandora/object/mathos:441. Acesso em: 21 ago. 2024.

SEVERINO, A. J. **Metodologia do Trabalho científico.** São Paulo: Cortez Editora, 2017. ISBN 978- 8524925207. Disponível em: https://plataforma.bvirtual.com.br. Acesso em: 24 jun. 2024.

SHKODRA, Endrit; JAJAGA, Edmond; SHALA, Mehmet. **Development and performance analysis of RESTful APIs in Core and Node.js using MongoDB database.** *In*: Proceedings of the 17th International Conference on Web Information Systems and Technologies (WEBIST 2021), SciTePress, p. 227-234, 2021. Disponível em: https://www.scitepress.org/Papers/2021/106212/106212.pdf. Acesso em: 18 set. 2024.

SOUZA, Estêvão Henrique Cangussú de et al. **Estudo comparativo de desempenho entre API desenvolvida com Spring WebFlux e Node.js.** II Worktec - Workshop de Tecnologia da Fatec Ribeirão Preto, [S. l.], v. 1, p. 1-2, 3 ago. 2020. Disponível em: http://www.fatecrp.edu.br/WorkTec/edicoes/2020-2/index.html. Acesso em: 13 ago. 2024.

SRIVASTAVA, Anushka. **CRUD API.** 2023. Trabalho de Conclusão de Curso (Bacharelado em Engenharia da Computação) – Jaypee University of Information Technology, Himachal Pradesh, 2023. Disponível em: http://www.ir.juit.ac.in:8080/jspui/handle/123456789/9864. Acesso em: 17 set. 2024.

TELLES, Diego. **Princípios de uma API REST.** 2023. Disponível em: https://unicorncoder.medium.com/princ%C3%ADpios-de-uma-api-rest-c8e08c2ba331. Acesso em: 18 ago. 2024.

TROCHIM, W.; DONNELLY, J. P. **The Research Methods Knowledge Base:** Types of designs. 2020. WebRef. Disponível em: https://faculty.cengage.com/titles/9781133954774. Acesso em: 25 jun. 2024.

Yin, R. K. **Estudo de caso:** planejamento e métodos. Syria Studies, v. 7, n. 1, 2004. Disponível em: http://maratavarespsictics.pbworks.com/w/file/fetch/74304716/3-YIN-planejamento_metodologia.pdf. Acesso em: 25 jun. 2024.

APÊNDICES

APÊNDICE A - Scripts de testes do cenário POST

A.1 - Script de teste do cenário POST para 10 usuários simultâneos

```
1   import http from 'k6/http';
2   import { check, sleep } from 'k6';
3
4   export let options = {
5     stages: [
6       { duration: '3m', target: 10 },
7     ],
8   };
9
10  export default function () {
11    const url = 'http://localhost:7000/books';
12
13    const uniqueTitle = `Livro-PNPM-10-${__VU}-${__ITER}`;
14
15    const payload = JSON.stringify({
16      title: uniqueTitle,
17      author: 'Autor',
18      year: 2005,
19      numberPages: 193,
20      genreId: '82b7f9a6-687a-4b0c-a781-6e414f3ecc90',
21    });
22
23    const params = {
24      headers: {
25        'Content-Type': 'application/json',
26      },
27    };
28
29    const res = http.post(url, payload, params);
30
31    const success = check(res, {
32      'status é 201': (r) => r.status === 201,
33    });
34
35    sleep(1);
36  }
```

A.2 - Script de teste do cenário POST para 100 usuários simultâneos

```
import http from 'k6/http';
import { check, sleep } from 'k6';

export let options = {
  stages: [
    { duration: '3m', target: 100 },
  ],
};

export default function () {
  const url = 'http://localhost:7000/books';

  const uniqueTitle = `Livro-PNPM-100-${__VU}-${__ITER}`;

  const payload = JSON.stringify({
    title: uniqueTitle,
    author: 'Autor',
    year: 2005,
    numberPages: 193,
    genreId: '256eeb0b-dfad-4855-ab97-18515c42c46a',
  });

  const params = {
    headers: {
      'Content-Type': 'application/json',
    },
  };

  const res = http.post(url, payload, params);

  const success = check(res, {
    'status é 201': (r) => r.status === 201,
  });

  sleep(1);
}
```

A.3 - Script de teste do cenário POST para 500 usuários simultâneos

```
1  import http from 'k6/http';
2  import { check, sleep } from 'k6';
3
4  export let options = {
5    stages: [
6      { duration: '3m', target: 500 },
7    ],
8  };
9
10 export default function () {
11   const url = 'http://localhost:7000/books';
12
13   const uniqueTitle = `Livro-PNPM-500-${__VU}-${__ITER}`;
14
15   const payload = JSON.stringify({
16     title: uniqueTitle,
17     author: 'Autor',
18     year: 2005,
19     numberPages: 193,
20     genreId: '256eeb0b-dfad-4855-ab97-18515c42c46a',
21   });
22
23   const params = {
24     headers: {
25       'Content-Type': 'application/json',
26     },
27   };
28
29   const res = http.post(url, payload, params);
30
31   const success = check(res, {
32     'status é 201': (r) => r.status === 201,
33   });
34
35   sleep(1);
36 }
```

A.4 - Script de teste do cenário POST para 1000 usuários simultâneos

```
import http from 'k6/http';
import { check, sleep } from 'k6';

export let options = {
  stages: [
    { duration: '3m', target: 1000 },
  ],
};

export default function () {
  const url = 'http://localhost:7000/books';

  const uniqueTitle = `Livro-PNPM-1000-${__VU}-${__ITER}`;

  const payload = JSON.stringify({
    title: uniqueTitle,
    author: 'Autor',
    year: 2005,
    numberPages: 193,
    genreId: '256eeb0b-dfad-4855-ab97-18515c42c46a',
  });

  const params = {
    headers: {
      'Content-Type': 'application/json',
    },
  };

  const res = http.post(url, payload, params);

  const success = check(res, {
    'status é 201': (r) => r.status === 201,
  });

  sleep(1);
}
```

APÊNDICE B - Scripts de testes do cenário GET

B.1 - Script de teste do cenário GET para 10 usuários simultâneos

```
1  import http from 'k6/http';
2  import { check, sleep } from 'k6';
3
4  export let options = {
5    stages: [
6      { duration: '3m', target: 10 },
7    ],
8  };
9
10 export default function () {
11   const url = 'http://localhost:7000/books';
12
13   const res = http.get(url);
14
15   const success = check(res, {
16     'status é 200': (r) => r.status === 200,
17   });
18
19   sleep(1);
20 }
```

B.2 - Script de teste do cenário GET para 100 usuários simultâneos

```
1  import http from 'k6/http';
2  import { check, sleep } from 'k6';
3
4  export let options = {
5    stages: [
6      { duration: '3m', target: 100 },
7    ],
8  };
9
10 export default function () {
11   const url = 'http://localhost:7000/books';
12
13   const res = http.get(url);
14
15   const success = check(res, {
16     'status é 200': (r) => r.status === 200,
17   });
18
19   sleep(1);
20 }
```

B.3 - Script de teste do cenário GET para 500 usuários simultâneos

```
import http from 'k6/http';
import { check, sleep } from 'k6';

export let options = {
  stages: [
    { duration: '3m', target: 500 },
  ],
};

export default function () {
  const url = 'http://localhost:7000/books';

  const res = http.get(url);

  const success = check(res, {
    'status é 200': (r) => r.status === 200,
  });

  sleep(1);
}
```

B.4 - Script de teste do cenário GET para 1000 usuários simultâneos

```
import http from 'k6/http';
import { check, sleep } from 'k6';

export let options = {
  stages: [
    { duration: '3m', target: 1000 },
  ],
};

export default function () {
  const url = 'http://localhost:7000/books';

  const res = http.get(url);

  const success = check(res, {
    'status é 200': (r) => r.status === 200,
  });

  sleep(1);
}
```

APÊNDICE C - Scripts de testes do cenário PUT

C.1 - Script de teste do cenário PUT para 10 usuários simultâneos

```
import http from 'k6/http';
import { check, group, sleep } from 'k6';

let resourceIds = [];
let currentIndex = 0;

function getResourceIds() {
  const getResponse = http.get('http://localhost:7000/books');
  check(getResponse, {
    'GET request is successful': (r) => r.status === 200,
  });

  const resources = getResponse.json();
  if (resources.length > 0) {
    return resources.map(resource => resource.id);
  } else {
    console.error('Nenhum recurso encontrado para atualização.');
    return [];
  }
}

export function setup() {
  resourceIds = getResourceIds();
  return { resourceIds };
}

export const options = {
  stages: [
    { duration: '3m', target: 10 },
  ],
};

export default function (data) {
  const { resourceIds } = data;

  group('PUT book', function () {
    const idToUse = resourceIds[currentIndex];

    currentIndex = (currentIndex + 1) % resourceIds.length;

    const uniqueTitle = `Book-PNPM-10-${__VU}-${__ITER}`;

    const payload = JSON.stringify({
      title: uniqueTitle,
      author: 'Author',
      year: 2015,
      numberPages: 57,
      genreId: '256eeb0b-dfad-4855-ab97-18515c42c46a',
    });

    const params = {
      headers: {
        'Content-Type': 'application/json',
      },
    };

    const putResponse = http.put(`http://localhost:7000/books/${idToUse}`, payload, params);

    check(putResponse, {
      'PUT request is successful': (r) => r.status === 200,
    });

    sleep(1);
  });
}
```

C.2 - Script de teste do cenário PUT para 100 usuários simultâneos

```
1  import http from 'k6/http';
2  import { check, group, sleep } from 'k6';
3
4  let resourceIds = [];
5  let currentIndex = 0;
6
7  function getResourceIds() {
8    const getResponse = http.get('http://localhost:7000/books');
9    check(getResponse, {
10     'GET request is successful': (r) => r.status === 200,
11   });
12
13   const resources = getResponse.json();
14   if (resources.length > 0) {
15     return resources.map(resource => resource.id);
16   } else {
17     console.error('Nenhum recurso encontrado para atualização.');
18     return [];
19   }
20 }
21
22 export function setup() {
23   resourceIds = getResourceIds();
24   return { resourceIds };
25 }
26
27 export const options = {
28   stages: [
29     { duration: '3m', target: 100 },
30   ],
31 };
32
33 export default function (data) {
34   const { resourceIds } = data;
35
36   group('PUT book', function () {
37     const idToUse = resourceIds[currentIndex];
38
39     currentIndex = (currentIndex + 1) % resourceIds.length;
40
41     const uniqueTitle = `Book-PNPM-100-${__VU}-${__ITER}`;
42
43     const payload = JSON.stringify({
44       title: uniqueTitle,
45       author: 'Author',
46       year: 2015,
47       numberPages: 57,
48       genreId: '256eeb0b-dfad-4855-ab97-18515c42c46a',
49     });
50
51     const params = {
52       headers: {
53         'Content-Type': 'application/json',
54       },
55     };
56
57     const putResponse = http.put(`http://localhost:7000/books/${idToUse}`, payload, params);
58
59     check(putResponse, {
60       'PUT request is successful': (r) => r.status === 200,
61     });
62
63     sleep(1);
64   });
65 }
```

C.3 - Script de teste do cenário PUT para 500 usuários simultâneos

```
1   import http from 'k6/http';
2   import { check, group, sleep } from 'k6';
3
4   let resourceIds = [];
5   let currentIndex = 0;
6
7   function getResourceIds() {
8     const getResponse = http.get('http://localhost:7000/books');
9     check(getResponse, {
10      'GET request is successful': (r) => r.status === 200,
11    });
12
13    const resources = getResponse.json();
14    if (resources.length > 0) {
15      return resources.map(resource => resource.id);
16    } else {
17      console.error('Nenhum recurso encontrado para atualização.');
18      return [];
19    }
20  }
21
22  export function setup() {
23    resourceIds = getResourceIds();
24    return { resourceIds };
25  }
26
27  export const options = {
28    stages: [
29      { duration: '3m', target: 500 },
30    ],
31  };
32
33  export default function (data) {
34    const { resourceIds } = data;
35
36    group('PUT book', function () {
37      const idToUse = resourceIds[currentIndex];
38
39      currentIndex = (currentIndex + 1) % resourceIds.length;
40
41      const uniqueTitle = `Book-PNPM-500-${__VU}-${__ITER}`;
42
43      const payload = JSON.stringify({
44        title: uniqueTitle,
45        author: 'Author',
46        year: 2015,
47        numberPages: 57,
48        genreId: '256eeb0b-dfad-4855-ab97-18515c42c46a',
49      });
50
51      const params = {
52        headers: {
53          'Content-Type': 'application/json',
54        },
55      };
56
57      const putResponse = http.put(`http://localhost:7000/books/${idToUse}`, payload, params);
58
59      check(putResponse, {
60        'PUT request is successful': (r) => r.status === 200,
61      });
62
63      sleep(1);
64    });
65  }
```

C.4 - Script de teste do cenário PUT para 1000 usuários simultâneos

```
import http from 'k6/http';
import { check, group, sleep } from 'k6';

let resourceIds = [];
let currentIndex = 0;

function getResourceIds() {
  const getResponse = http.get('http://localhost:7000/books');
  check(getResponse, {
    'GET request is successful': (r) => r.status === 200,
  });

  const resources = getResponse.json();
  if (resources.length > 0) {
    return resources.map(resource => resource.id);
  } else {
    console.error('Nenhum recurso encontrado para atualização.');
    return [];
  }
}

export function setup() {
  resourceIds = getResourceIds();
  return { resourceIds };
}

export const options = {
  stages: [
    { duration: '3m', target: 1000 },
  ],
};

export default function (data) {
  const { resourceIds } = data;

  group('PUT book', function () {
    const idToUse = resourceIds[currentIndex];

    currentIndex = (currentIndex + 1) % resourceIds.length;

    const uniqueTitle = `Book-PNPM-1000-${__VU}-${__ITER}`;

    const payload = JSON.stringify({
      title: uniqueTitle,
      author: 'Author',
      year: 2015,
      numberPages: 57,
      genreId: '256eeb0b-dfad-4855-ab97-18515c42c46a',
    });

    const params = {
      headers: {
        'Content-Type': 'application/json',
      },
    };

    const putResponse = http.put(`http://localhost:7000/books/${idToUse}`, payload, params);

    check(putResponse, {
      'PUT request is successful': (r) => r.status === 200,
    });

    sleep(1);
  });
}
```

APÊNDICE D - Scripts de testes do cenário DELETE

D.1 - Script de teste do cenário DELETE para 10 usuários simultâneos

```
1  import http from 'k6/http';
2  import { check, group, sleep } from 'k6';
3
4  let resourceIds = [];
5  let currentIndex = 0;
6
7  function getResourceIds() {
8    const getResponse = http.get('http://localhost:7000/books');
9    check(getResponse, {
10     'GET request is successful': (r) => r.status === 200,
11   });
12
13   const resources = getResponse.json();
14   if (resources.length > 0) {
15     return resources.map(resource => resource.id);
16   } else {
17     console.error('Nenhum recurso encontrado para exclusão.');
18     return [];
19   }
20 }
21
22 export function setup() {
23   resourceIds = getResourceIds();
24   return { resourceIds };
25 }
26
27 export const options = {
28   stages: [
29     { duration: '3m', target: 10 },
30   ],
31 };
32
33 export default function (data) {
34   const { resourceIds } = data;
35
36   group('DELETE book', function () {
37     const idToUse = resourceIds[currentIndex];
38
39     currentIndex = (currentIndex + 1) % resourceIds.length;
40
41     const deleteResponse = http.del(`http://localhost:7000/books/${idToUse}`);
42
43     check(deleteResponse, {
44       'DELETE request is successful': (r) => r.status === 204,
45     });
46
47     sleep(1);
48   });
49 }
```

D.2 - Script de teste do cenário DELETE para 100 usuários simultâneos

```javascript
import http from 'k6/http';
import { check, group, sleep } from 'k6';

let resourceIds = [];
let currentIndex = 0;

function getResourceIds() {
  const getResponse = http.get('http://localhost:7000/books');
  check(getResponse, {
    'GET request is successful': (r) => r.status === 200,
  });

  const resources = getResponse.json();
  if (resources.length > 0) {
    return resources.map(resource => resource.id);
  } else {
    console.error('Nenhum recurso encontrado para exclusão.');
    return [];
  }
}

export function setup() {
  resourceIds = getResourceIds();
  return { resourceIds };
}

export const options = {
  stages: [
    { duration: '3m', target: 100 },
  ],
};

export default function (data) {
  const { resourceIds } = data;

  group('DELETE book', function () {
    const idToUse = resourceIds[currentIndex];

    currentIndex = (currentIndex + 1) % resourceIds.length;

    const deleteResponse = http.del(`http://localhost:7000/books/${idToUse}`);

    check(deleteResponse, {
      'DELETE request is successful': (r) => r.status === 204,
    });

    sleep(1);
  });
}
```

D.3 - Script de teste do cenário DELETE para 500 usuários simultâneos

```javascript
import http from 'k6/http';
import { check, group, sleep } from 'k6';

let resourceIds = [];
let currentIndex = 0;

function getResourceIds() {
  const getResponse = http.get('http://localhost:7000/books');
  check(getResponse, {
    'GET request is successful': (r) => r.status === 200,
  });

  const resources = getResponse.json();
  if (resources.length > 0) {
    return resources.map(resource => resource.id);
  } else {
    console.error('Nenhum recurso encontrado para exclusão.');
    return [];
  }
}

export function setup() {
  resourceIds = getResourceIds();
  return { resourceIds };
}

export const options = {
  stages: [
    { duration: '3m', target: 500 },
  ],
};

export default function (data) {
  const { resourceIds } = data;

  group('DELETE book', function () {
    const idToUse = resourceIds[currentIndex];

    currentIndex = (currentIndex + 1) % resourceIds.length;

    const deleteResponse = http.del(`http://localhost:7000/books/${idToUse}`);

    check(deleteResponse, {
      'DELETE request is successful': (r) => r.status === 204,
    });

    sleep(1);
  });
}
```

D.4 - Script de teste do cenário DELETE para 1000 usuários simultâneos

```javascript
import http from 'k6/http';
import { check, group, sleep } from 'k6';

let resourceIds = [];
let currentIndex = 0;

function getResourceIds() {
  const getResponse = http.get('http://localhost:7000/books');
  check(getResponse, {
    'GET request is successful': (r) => r.status === 200,
  });

  const resources = getResponse.json();
  if (resources.length > 0) {
    return resources.map(resource => resource.id);
  } else {
    console.error('Nenhum recurso encontrado para exclusão.');
    return [];
  }
}

export function setup() {
  resourceIds = getResourceIds();
  return { resourceIds };
}

export const options = {
  stages: [
    { duration: '3m', target: 1000 },
  ],
};

export default function (data) {
  const { resourceIds } = data;

  group('DELETE book', function () {
    const idToUse = resourceIds[currentIndex];

    currentIndex = (currentIndex + 1) % resourceIds.length;

    const deleteResponse = http.del(`http://localhost:7000/books/${idToUse}`);

    check(deleteResponse, {
      'DELETE request is successful': (r) => r.status === 204,
    });

    sleep(1);
  });
}
```