UNIVERSIDADE DO VALE DO TAQUARI - UNIVATES

CURSO DE ENGENHARIA DE SOFTWARE

USO DE RATE LIMIT PARA GARANTIR A ESTABILIDADE E DISPONIBILIDADE DE APIS

Rafael Luiz Siebeneichler

Lajeado/RS, dezembro de 2024

Rafael Luiz Siebeneichler

USO DE RATE LIMIT PARA GARANTIR A ESTABILIDADE E DISPONIBILIDADE DE APIS

Monografia apresentada no componente curricular Trabalho de Conclusão de Curso - Etapa II, do curso de Engenharia de Software, da Universidade do Vale do Taquari - Univates, como parte da exigência para a obtenção do título de Bacharel em Engenharia de Software.

Orientador: Prof. Dr. Alexandre Sturmer Wolf

Lajeado/RS, dezembro de 2024

Rafael Luiz Siebeneichler

USO DE RATE LIMIT PARA GARANTIR A ESTABILIDADE E DISPONIBILIDADE DE APIS

A Banca examinadora abaixo aprova a Monografia apresentada no componente curricular de Trabalho de Conclusão de Curso - Etapa II, do Curso de Graduação em Engenharia de Software, da Universidade do Vale do Taquari - Univates, como parte da exigência para a obtenção do título de Bacharel em Engenharia de Software:

Prof. Dr. Alexandre Sturmer Wolf – orientador
Universidade do Vale do Taquari - Univates

Prof. Me. Juliano Dertzbacher
Universidade do Vale do Taquari - Univates

Prof. Dr. Mouriac Halen Diemer
Universidade do Vale do Taquari - Univates

Lajeado/RS, dezembro de 2024

RESUMO

Em um cenário onde a disponibilidade e estabilidade de APIs são frequentemente comprometidas devido ao mau uso, abusos ou ataques de negação de serviço (DDoS), o *rate limit* emerge como uma solução eficaz para mitigar esses riscos. Conforme documentado por Toulas (2021), o uso de APIs tem crescido significativamente nos últimos anos, o que, por sua vez, levou a um aumento desproporcional nos ataques a essas interfaces, com um crescimento de 600% em 2021. Esses ataques não apenas comprometem a segurança das APIs, mas também afetam diretamente sua estabilidade e disponibilidade, ressaltando a necessidade urgente de estratégias de segurança robustas. A governança de APIs, através da aplicação de padrões de *design* como o *rate limit*, desempenha um papel crucial na proteção dessas interfaces. Ao limitar o número de solicitações permitidas por unidade de tempo, pode-se prevenir sobrecargas, mitigar o impacto de acessos abusivos e proteger contra ataques direcionados. Além disso, essa técnica garante uma distribuição justa dos recursos entre os usuários, o que é essencial para manter a qualidade do serviço, especialmente em ambientes de alta demanda. O desenvolvimento de uma API e a realização de testes no presente estudo de caso validam a eficácia dessa abordagem. No cenário sem *rate limit*, de cerca de 50.000 requisições em um curto período de tempo, 82,84% falham devido à sobrecarga do sistema. A latência média das requisições de sucesso é de 13 segundos, e o percentil 90 (p90) alcança 26,76 segundos, evidenciando a instabilidade do sistema sob alta carga. Por outro lado, no cenário com *rate limit*, os erros observados são exclusivamente causados pelo bloqueio da técnica, indicando que os abusos foram controlados conforme esperado pelas políticas configuradas. Assim, a latência das requisições bem-sucedidas se manteve em milissegundos, destacando a estabilidade proporcionada pela solução. Portanto, a adoção de práticas como o *rate limit* não só promove a resiliência das APIs contra falhas e incidentes de segurança, mas também contribui para o desenvolvimento de aplicações mais confiáveis e robustas, assegurando que essas APIs possam operar de maneira estável e segura mesmo sob condições adversas.

Palavras-chave: Web Services; API; Rate Limit; Disponibilidade; Teste de carga;

ABSTRACT

In a scenario where the availability and stability of APIs are often compromised due to misuse, abuse or denial of service (DDoS) attacks, rate limiting emerges as an effective solution to mitigate these risks. As documented by Toulas (2021), the use of APIs has grown significantly in recent years, which in turn has led to a disproportionate increase in attacks on these interfaces, with a growth of 600% in 2021. These attacks not only compromise the security of APIs, but also directly affect their stability and availability, highlighting the urgent need for robust security strategies. API governance, through the application of design patterns such as rate limit, plays a crucial role in protecting these interfaces. By limiting the number of requests allowed per unit of time, you can prevent overloads, mitigate the impact of abusive access and protect against targeted attacks. In addition, this technique ensures a fair distribution of resources among users, which is essential for maintaining quality of service, especially in high-demand environments. The development of an API and testing carried out in this case study validate the effectiveness of this approach. In the scenario without rate limiting, of around 50,000 requests in a short period of time, 82.84% fail due to system overload. The average latency of successful requests is 13 seconds, and the 90th percentile (p90) is 26.76 seconds, showing the instability of the system under high load. On the other hand, in the scenario with rate limit, the errors observed are exclusively caused by the blocking of the technique, indicating that abuse was controlled as expected by the policies configured. Thus, the latency of successful requests remained in milliseconds, highlighting the stability provided by the solution. Therefore, adopting practices such as rate limiting not only promotes API resilience against failures and security incidents, but also contributes to the development of more reliable and robust applications, ensuring that these APIs can operate in a stable and secure manner even under adverse conditions.

Keywords: Web Services; API; Rate Limit; Availability; Load Testing;

LISTA DE FIGURAS

Figura 1 – Uso de APIs em uma arquitetura moderna baseada em nuvem.……..…22
Figura 2 – Rate Limit: Quando o cliente excede o número permitido de pedidos por período de tempo, todos os outros pedidos são recusados………………...………29
Figura 3 – Fluxograma da arquitetura do framework de Ströher…..…………..………37
Figura 4 – Reações de APIs ao exceder o limite de taxa de requisições……………38
Figura 5 – Diagrama de caso de uso…………………………………………..………50
Figura 6 – Diagrama da arquitetura da aplicação proposta……………………………52
Figura 7 – Código fonte da configuração do rate limit.…..……………………………..53
Figura 8 – Código fonte validação dos pontos de rate limit..…………………………..54
Figura 9 – Código fonte instrumentalização do container em Docker………………..55
Figura 10 – Código fonte migrations e seeders para geração da tabela e dados no banco de dados…………………………………………………………………………..56
Figura 11 – Requisição via Postman para rota com rate limit com sucesso….……..58
Figura 12 – Requisição via Postman para rota com rate limit ao exceder o limite configurado…………………………………..……………………………………..59
Figura 13 – Exemplo dos dados coletados pelo k6…...……...………………………..61
Figura 14 – Script utilizado para executar os testes de carga do k6 e coletar dados do banco de dados.……………………………………………………………………62
Figura 15 – Resultados do k6 nos testes de carga sem rate limit………...………..63
Figura 16 – Métricas de monitoramento do banco de dados nos testes de carga sem rate limit.….……………………………………………………………………………64
Figura 17 – Logs de erro ao na API e ao conectar no banco de dados………..……..64

Figura 18 – Resultados do k6 nos testes de carga com rate limit…………...………..65
Figura 19 – Métricas de monitoramento do banco de dados nos testes de carga com rate limit.…..……………………………………………………………………………66

LISTA DE QUADROS

Quadro 1 – Comparativo dos trabalhos relacionados……...……...…………………...39
Quadro 2 – Comparativo dos resultados dos testes……………….…………….........66

LISTA DE ABREVIATURAS E SIGLAS

API *Application Programming Interface* – Interface de Programação de Aplicações

CLI *Command-Line Interface* – Interface de Linha de Comando

CPU *Central Processing Unit* – Unidade Central de Processamento

DDoS *Distributed Denial of Service* – Negação de serviço distribuído

DoS *Denial of Service* – Negação de Serviço

HTML *Hypertext Markup Language* – Linguagem de Marcação de Hipertexto

HTTP *Hypertext Transfer Protocol* – Protocolo de Transferência de Hipertexto

IP *Internet Protocol* – Protocolo de Rede

JSON JavaScript Object Notation – Notação de Objetos JavaScript

JWT *JSON Web Token*

MFA *Multi-Factor Authentication* – Múltiplo Fator de Autenticação

ORM *Object-Relational Mapping* – Mapeamento Objeto-Relacional

REST *Representational State Transfer* – Transferência de Estado Representacional

RL Rate Limit – Limite de Taxa

SLA *Service Level Agreement* – Nível de Serviço Acordado

SLO *Service Level Objective* – Objetivo de Nível de Serviço

SOAP *Simple Object Access Protocol* – Protocolo Simples de Acesso a Objetos

SQL *Structured Query Language* – Linguagem de Consulta Estruturada

TCP *Transmission Control Protocol* – Protocolo de Controle de Transmissão

XML *Extensible Markup Language* – Linguagem de Marcação Extensível

SUMÁRIO

1 INTRODUÇÃO 10
1.1 Problema de pesquisa 12
1.2 Hipótese 12
1.3 Objetivo Geral 13
1.3.1 Objetivos Específicos 13
1.4 Justificativa 14
1.5 Estrutura do trabalho 14

2 FUNDAMENTAÇÃO TEÓRICA 16
2.1 Contexto histórico e evolução até APIs 16
2.1.1 World Wide Web 17
2.1.2 Protocolo HTTP 18
2.1.3 SOAP 19
2.1.4 REST 20
2.1.5 API 21
2.2 Segurança no contexto de APIs 23
2.2.1 Confidencialidade 23
2.2.2 Integridade 24
2.2.3 Disponibilidade 24
2.2.4 Rastreabilidade 25
2.2.5 Identificação e autenticação 25
2.2.6 Ameaças à APIs 27
2.2.6.1 Força Bruta 28
2.2.6.2 DDoS 28
2.3 Rate Limit 28
2.3.1 Acordos de nível de serviço 29
2.3.2 Nível de Objetivo de Serviço 30
2.3.3 Plano de precificação 31

2.3.4 Abordagens de rate limit 31
2.3.5 Algoritmos de rate limit 32
2.4 Testes 32

3 TRABALHOS RELACIONADOS 35
3.1 DataCache: Gestão e padronização na comunicação entre aplicações 35
3.2 Disponibilização de dados de fundos de investimento através de uma API 36
3.3 Framework para desenvolvimento e implantação de aplicações HTTP utilizando serverless computing 36
3.4 Impact of API Rate Limit on Reliability of Microservices-Based Architectures 37
3.5 API Rate Limit Adoption – A pattern collection 38
3.6 Comparativo entre os trabalhos relacionados 39

4 MATERIAIS E MÉTODOS 41
4.1 Pesquisa quanto aos Métodos Científicos 41
4.1.1 Pesquisa quanto ao Modo de abordagem 41
4.1.2 Pesquisa quanto aos Fins da Pesquisa 42
4.1.3 Pesquisa enquanto aos Procedimentos técnicos 42
4.2 Tecnologias 43
4.2.1 TypeScript 43
4.2.2 Node Package Manager (NPM) 44
4.2.3 NestJS 44
4.2.4 Grafana k6 45
4.2.5 Docker 46
4.2.6 Sequelize 46
4.2.7 PostgreSQL 47
4.2.8 Postman 48
4.3 Desenvolvimento 48
4.3.1 Escopo 49
4.3.2 Arquitetura 50
4.3.2 Rate Limit 52
4.3.3 Banco de dados e geração dos dados 54

5 TESTES E ANÁLISE DOS RESULTADOS 58
5.1 Validação do Sistema de Rate Limit 58
5.1.1 Limitação de requisições por unidade de tempo 58
5.1.2 Resposta do sistema ao exceder o limite de taxa 59
5.2 Cenários de Testes de Carga 60
5.3 Processo de Coleta dos Dados 61

5.3.1 Utilização do k6 62
5.3.1 Ferramentas auxiliares 63
5.4 Análise dos resultados obtidos 64
5.4.1 Resultados sem rate limit 64
5.4.2 Resultados com rate limit 66
5.4.3 Sumarização dos resultados 67

6 CONSIDERAÇÕES FINAIS 69

REFERÊNCIAS 72

1 INTRODUÇÃO

Sommerville (2011) define a engenharia de software como uma disciplina que abrange todos os aspectos da produção de software, desde os estágios iniciais de especificação até a manutenção após a entrega do sistema. Ele destaca que um dos maiores desafios da engenharia de software é garantir a qualidade, a disponibilidade e a estabilidade dos sistemas desenvolvidos, especialmente em ambientes cada vez mais complexos e dinâmicos. Esses desafios se tornam ainda mais pronunciados quando o software é construído sem a aplicação rigorosa das técnicas e métodos estabelecidos pela engenharia de software, comprometendo a eficácia e a confiabilidade dos sistemas.

Por um lado, enfrentamos os desafios de construir softwares com qualidade, por outro, lidamos com um ambiente cada vez mais hostil, que busca explorar vulnerabilidades em nossos sistemas. Conforme destacado por Bill Toulas (2021), o uso de APIs (Application Programming Interfaces) tem crescido exponencialmente nos últimos anos, impulsionado pela necessidade de integração entre diferentes sistemas e pela criação de ecossistemas mais dinâmicos e interconectados. No entanto, Toulas observa que esse aumento no uso de APIs foi acompanhado por um crescimento desproporcional de ataques direcionados a essas interfaces. Especificamente, ele aponta que, em 2021, os ataques abusando de APIs cresceram mais de 600%, evidenciando a tentativa de explorar vulnerabilidades dessas interfaces.

Toulas (2021) também ressalta que, apesar do crescimento significativo no uso de APIs, muitas organizações ainda não adotaram estratégias de segurança robustas para protegê-las. Essa falta de medidas de segurança deixa as APIs expostas a diversos riscos, como o uso indevido e os ataques de negação de

serviço (*Denial of Service* - DoS). Esses riscos podem comprometer a disponibilidade e a estabilidade das APIs; de acordo com a pesquisa mencionada por Toulas, os riscos mencionados acima representam 33% das maiores preocupações em relação à segurança de APIs. Isso ressalta a necessidade de implementar estratégias de segurança mais eficazes para mitigar os riscos crescentes associados ao uso de APIs.

Dentre as técnicas, métodos e processos de engenharia de software para prevenir e mitigar os riscos comentados acima surgem os padrões de design de APIs que são fundamentais para mitigar os riscos associados ao desenvolvimento e uso de APIs. Conforme destacado por Zimmermann et al. (2023), a padronização no design de APIs não apenas promove a consistência e a previsibilidade das interfaces, mas também desempenha um papel crucial na segurança e na resiliência das aplicações que dependem dessas APIs. Ao seguir padrões bem estabelecidos, as equipes de desenvolvimento podem garantir que as APIs sejam robustas, fáceis de integrar e que possuam mecanismos de defesa eficazes contra ataques e abusos.

Entre os métodos recomendados, estão a implementação de contratos de API claros, a utilização de versionamento adequado e a adoção de práticas de autenticação e autorização robustas. Esses padrões ajudam a proteger as APIs contra vulnerabilidades, garantindo que cada chamada de API seja validada e processada de acordo com regras bem definidas, minimizando o risco de exploits. Além disso, técnicas como o uso de *rate limit* (RL), ou, em português, limite de taxa, e a monitoração contínua, que integram as boas práticas de design de APIs, são fundamentais para prevenir sobrecargas e garantir a estabilidade do sistema, mesmo em condições adversas. Dessa forma, os padrões de design de APIs se tornam indispensáveis para garantir a qualidade e a segurança dos serviços expostos na web, proporcionando uma camada adicional de proteção contra as ameaças emergentes no cenário digital.

Diante desse contexto desafiador e da busca por soluções eficazes, o presente trabalho se propõe a explorar a aplicação do padrão de design de *rate limit* como uma estratégia eficaz para promover a estabilidade e a resiliência dessas interfaces. A técnica busca equilibrar o uso dos recursos, prevenindo abusos e

garantindo uma distribuição justa entre os usuários. Este estudo contribui para a identificação de práticas que asseguram o funcionamento confiável das APIs, especialmente em cenários de alta demanda, onde a sobrecarga pode comprometer tanto a experiência do usuário quanto a integridade dos sistemas.

1.1 Problema de pesquisa

O tema do presente trabalho busca pesquisar, implementar e testar uma API com e sem a utilização de tecnologias de *RL*. A pesquisa busca validar a utilização de limitação de taxas de requisições com testes de cargas e identificar as melhores práticas da literatura e do mercado de trabalho.

Tendo em vista que Web services e APIs podem ser seriamente afetados por má utilização, abusos, ou ataques de negação de serviço (DDoS) e esses ataques sobrecarregam os sistemas com tráfego malicioso, levando a interrupções e falhas significativas na prestação de serviços. Comprometendo a disponibilidade, a estabilidade e a confiabilidade das aplicações. Como documentado em casos famosos pela Cloudflare (2024), ataques de grande escala podem paralisar redes inteiras, interrompendo a comunicação entre servidores e clientes, e causando um impacto devastador na infraestrutura digital.

Desta forma, como problema de pesquisa busca-se investigar e compreender de que maneira a utilização de *rate limit* contribui para a estabilidade e proteção de APIs?

1.2 Hipótese

A aplicação de *rate limit* como parte da governança de APIs desempenha um papel importante na manutenção da estabilidade e disponibilidade dos serviços expostos. Ao limitar o número de requisições permitidas por usuário e unidade de tempo, essa técnica tem o potencial de auxiliar na prevenção de sobrecargas e na mitigação dos impactos de acessos abusivos e ataques direcionados, especialmente em situações de alta demanda e utilização excessiva. Além disto, acredita-se que esta técnica possa garantir uma distribuição mais justa dos recursos entre os

usuários. Dessa forma, espera-se que contribua para o aumento da resiliência das APIs contra falhas e incidentes de segurança.

1.3 Objetivo Geral

Este trabalho tem como objetivo principal investigar e demonstrar como a técnica de *rate limit* pode ser utilizada como uma ferramenta eficaz de governança para garantir a estabilidade e disponibilidade de APIs. O estudo visa identificar os benefícios, desafios e melhores práticas associadas à implementação desta técnica, com foco em como essa técnica pode prevenir sobrecargas, proteger contra abusos e assegurar a distribuição equitativa de recursos em um ambiente de API. Ao final, espera-se fornecer uma compreensão aprofundada de como o *rate limit* contribui para a robustez e confiabilidade de serviços que dependem de APIs.

1.3.1 Objetivos Específicos

Com a finalidade de atingir o objetivo geral, foram estabelecidos os seguintes objetivos específicos:

- Analisar os conceitos fundamentais de governança de APIs e como o *rate limit* se integra a esse contexto;
- Explorar diferentes tipos de algoritmos de *rate limit*, avaliando suas características, vantagens e desvantagens em cenários práticos;
- Investigar as implicações do *rate limit* na estabilidade e disponibilidade de APIs, incluindo estudos de caso e exemplos reais de aplicação;
- Analisar a aplicação prática de *rate limit* em uma API, detalhando os aspectos do processo de configuração, monitoramento e ajustes necessários para sua aplicação eficaz;
- Examinar os desafios associados à adoção de *rate limit*, propondo soluções e boas práticas para otimizar sua eficácia em ambientes de produção.

1.4 Justificativa

A crescente utilização de APIs como elementos cruciais na integração de sistemas e na criação de ecossistemas digitais dinâmicos tem trazido desafios significativos relacionados à sua segurança e estabilidade. O aumento exponencial de ataques direcionados a APIs, com um crescimento de 600% em 2021, destaca a urgência em adotar medidas eficazes para proteger esses componentes de software. (TOULAS, 2022).

Neste contexto, o uso de *rate limit* se apresenta como uma ferramenta essencial para governança de APIs, garantindo que a disponibilidade e estabilidade sejam mantidas, mesmo sob alta demanda. Ao limitar a quantidade de solicitações permitidas em um determinado período, ele não apenas protege contra abusos e sobrecargas, mas também assegura uma distribuição justa dos recursos, promovendo o desenvolvimento de aplicações mais confiáveis e robustas. Assim, este trabalho busca investigar como a implementação dessa técnica pode contribuir para a qualidade e a resiliência de serviços que dependem de APIs.

1.5 Estrutura do trabalho

O presente trabalho está organizado em capítulos que visam proporcionar uma estrutura coerente e abrangente. A Introdução apresenta o tema, a delimitação do escopo, o problema de pesquisa, e também estabelece a hipótese, o objetivo geral, os objetivos específicos do estudo, além da justificativa, que ressalta a importância e a relevância do trabalho. Além disso, a introdução inclui uma descrição da estrutura do trabalho, delineando a organização dos capítulos subsequentes.

O capítulo de Fundamentação Teórica oferece o embasamento conceitual e teórico relacionado ao tema em questão, explorando tópicos relevantes que contextualizam o problema e fornecem uma base de conhecimento para a pesquisa. No capítulo de Trabalhos Relacionados, são apresentados estudos anteriores relevantes sobre APIs, desde conceitos, implementações de API e padrões de

design de API. Ao final desse capítulo, é realizada uma breve comparação entre esses trabalhos e o presente estudo.

O capítulo de Materiais e Métodos descreve a caracterização da pesquisa, aborda também as tecnologias empregadas, como TypeScript, NPM, NestJS e Grafana k6. Além disso, discute o escopo do trabalho, a arquitetura do sistema e o desenvolvimento da aplicação. O próximo capítulo, Testes e Análises dos Resultados, descreve os cenários dos testes, os processos de coleta de dados, os resultados obtidos e a análise deles. Em seguida, o capítulo de Considerações Finais oferece conclusões com base nas validações realizadas ao longo do trabalho e sugestões de melhorias para trabalhos futuros. Por fim, o capítulo de Referências compila as fontes utilizadas para fundamentar e referenciar o trabalho, proporcionando uma base sólida para o conteúdo apresentado.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo são apresentados os conceitos-chaves adotados. A primeira subseção explica o que são APIs, desde sua origem até os benefícios de utilizá-las e suas principais características. A segunda subseção explora os tipos de ataques que uma API pode sofrer e como o *rate limit* auxilia na prevenção ou mitigação desses riscos. A terceira subseção trata do conceito de *rate limit*, as diferentes formas de aplicá-lo e os seus principais algoritmos. Por fim, a última subseção discorre sobre testes de softwares e alguns dos tipos de testes, destacando sua utilidade para validar a hipótese apresentada.

2.1 Contexto histórico e evolução até APIs

Em pleno século XXI, em um cenário de avanços tecnológicos exponenciais, compreender a origem de ferramentas e métodos, como as APIs, é essencial para contextualizar sua relevância no ecossistema digital. Embora frequentemente ignorado, o caminho histórico percorrido pelas APIs demonstra sua importância como "heróis silenciosos" que moldaram o cenário digital moderno, conectando sistemas de forma fluida e eficiente.

Desde os primórdios da computação, entre as décadas de 1950 e 1960, as APIs apresentavam suas raízes. Conforme registrado por Wilkes et al. (1951), o uso de bibliotecas surgiu como uma solução prática para atender à crescente necessidade de reutilizar código e simplificar o processo de programação. Na época, programadores perceberam que muitas funções e rotinas eram frequentemente repetidas, levando à criação de bibliotecas padronizadas que encapsulam essas

operações. Esse conceito de modularidade e reutilização foi pioneiro, introduzindo princípios que ainda hoje sustentam o desenvolvimento de APIs.

Em síntese, programadores ao escreverem rotinas e funções de uso geral, as reuniam em bibliotecas para facilitar projetos futuros e compartilhá-las com outros desenvolvedores. Essas bibliotecas tornam-se recursos valiosos, fornecendo códigos previamente testados, confiáveis e otimizados, que economizam tempo e esforço no desenvolvimento. Esse procedimento estabelece as bases para o que se conhece como o desenvolvimento moderno de APIs.

2.1.1 World Wide Web

A história das APIs é viabilizada com o advento da World Wide Web (WWW), a rede mundial de computadores, que revoluciona a comunicação global e possibilita novas formas de interação entre pessoas, como escrita, áudio, imagens e voz.

Essa trajetória inicia-se em 1945, quando o engenheiro norte-americano Vannevar Bush apresenta o conceito do Memex (Memory Extension) em seu artigo intitulado As We May Think (BUSH, 1945). O Memex consiste em uma máquina projetada para armazenar e recuperar grandes volumes de conhecimento, uma necessidade urgente da época devido ao crescimento exponencial da produção de informações sem métodos eficientes de organização e acesso. Operando por associações, o Memex é considerado o precursor do hipertexto, que mais tarde se torna um dos pilares da web.

Nos anos seguintes, a IBM introduz o SGML (Standard Generalized Markup Language), uma tecnologia que permite o processamento estruturado de informações, influenciando diretamente linguagens como o HTML. Na década de 1960, o visionário Ted Nelson idealizou o Projeto Xanadu, concebendo uma rede de computadores com interfaces simples e acessíveis para os usuários. Esse projeto introduz conceitos fundamentais para o desenvolvimento da internet contemporânea (WHITEHEAD, 2007).

A World Wide Web foi concluída no final de 1990, no CERN (*European Organization for Nuclear Research*), com a primeira página web tornando-se funcional em agosto de 1991. A partir desse momento, a WWW cresce

exponencialmente, alcançando um número estimado de 40 milhões de usuários em apenas cinco anos. Em determinados períodos, o número de usuários chega a dobrar a cada dois meses. Esse crescimento acelerado expõe a Web a riscos de colapso, devido à ausência de protocolos padronizados, suporte eficiente para cache e outras ferramentas estabilizadoras que garantam sua sustentabilidade e desempenho (MASSÉ, 2012).

Durante esse processo, surgiram diversos desafios, dentre eles como conectar e integrar diversos sistemas globalmente de forma eficiente. Nesse contexto, desenvolve-se o Hypertext Transfer Protocol (HTTP), um protocolo que viabiliza a comunicação entre servidores web e outros programas externos, permitindo o compartilhamento de dados. Esse avanço marca o surgimento das primeiras APIs web, que possibilitam a criação de conteúdo dinâmico e promovem a interatividade, abrindo caminho para o crescimento da internet.

2.1.2 Protocolo HTTP

O HTTP, é um dos principais protocolos existentes na internet, sendo utilizado para a distribuição de objetos de hipermídia referenciados por uma URI, baseando-se na realização de requisições por um cliente e no envio de respostas pelo servidor. Uma vez, tratando-se de um meio de comunicação, se mostra fundamental a determinação de certas regras para garantir a troca de informações entre servidor e cliente de forma padronizada (BERNERS-LEE et al., 1996).

Dessa maneira, uma requisição no protocolo HTTP apresenta dois elementos principais: o cabeçalho (*header*), que traz os metadados da requisição, como o código de status (*status code*), o formato do corpo da requisição e os parâmetros de autenticação, e o corpo (*body*), que contém os dados da requisição. Nesse sentido, sendo o cabeçalho e o corpo da requisição definidos conforme o esperado pelo serviço, também existe a necessidade de especificar o método da requisição, para que assim, que define qual o tipo de ação a ser executada pelo servidor (MASSÉ, 2011).

Entre os diversos métodos disponíveis, destacam-se quatro amplamente utilizados: GET, POST, PUT e DELETE (Massé, 2011). O método GET é utilizado

para recuperação de recursos existentes, sem alterar ou criar novas informações. O método POST, por sua vez, realiza a criação de novos recursos no servidor. O método PUT é empregado para atualizar recursos existentes, enquanto o método DELETE serve para excluir fisicamente ou logicamente informações no servidor.

Conforme detalhado por Fielding et al. (2022), as respostas do protocolo HTTP, são associadas a um código de status, que possui como objetivo, identificar se a requisição foi feita com sucesso, ou se houve algum erro sistêmico. Requisições com sucesso geralmente são associadas com códigos de 200 a 299, enquanto que requisições de 400 a 499 estão associadas com erros do lado do cliente, como por exemplo, falta do envio de algum parâmetro obrigatório, tentativa de acessar uma rota inexistente, falta de permissão para acessar o determinado recurso, entre outras situações. Por fim, códigos de status de 500 a 599 se referem a erros do lado do servidor, alguns exemplos bem recorrentes como erro interno do servidor (*internal server error*) ou serviço indisponível.

2.1.3 SOAP

Com a evolução da web, tornou-se cada vez mais evidente a necessidade de padrões robustos para comunicação entre aplicações. Essa demanda resultou na criação do *Simple Object Access Protocol* (SOAP) no final da década de 1990. O SOAP pode ser comparado a uma grande orquestra, onde cada nota é bem definida e todos seguem suas partituras de maneira rigorosa (DANTAS, 2007).

Desta forma, as APIs baseadas em SOAP ganharam popularidade por viabilizarem a troca de dados entre sistemas distintos, utilizando protocolos variados, como HTTP e TCP. Os padrões definidos do SOAP permitiam a criação de aplicações e serviços de forma fácil e coesa por seus desenvolvedores. Por isso, o SOAP rapidamente se consolida como uma ferramenta essencial no ambiente empresarial, funcionando como uma ponte entre sistemas de software heterogêneos.

No entanto, o SOAP enfrenta desafios que contribuem para seu desuso e a adoção de novos modelos, como o REST. Um dos principais problemas está na sua complexidade, já que a implementação e manutenção demandam grande esforço

devido às especificações detalhadas e ao uso obrigatório do XML para formatação de mensagens. Além disso, o desempenho do SOAP é prejudicado pelo peso de suas mensagens, que incluem metadados extensivos, tornando-o menos eficiente em redes com baixa largura de banda (HALILI; RAMADANI, 2018).

2.1.4 REST

Embora tenha sido amplamente utilizado, conforme comentado acima o SOAP é considerado pesado e complexo, características que o tornam menos adequado para a web moderna e aplicações móveis. Em resposta a essas limitações, surge o *Representational State Transfer* (REST), uma abordagem mais leve e simples que atende às necessidades da evolução tecnológica. O REST apresentou comandos bem estabelecidos, como GET, POST, PUT e DELETE, que possibilitaram interações eficientes e padronizadas entre aplicações e recursos web, aproveitando e popularizando métodos do protocolo HTTP abordados previamente. Nesse sentido, APIs RESTful, seguindo princípios REST, ganharam tração por conta de sua simplicidade e facilidade de uso (HALILI; RAMADANI, 2018).

Diferentemente do SOAP, o REST define uma forma de comunicação simples e eficiente entre os componentes da *web*, ou seja, como o REST não possui padrão fechado, existe a possibilidade de implementá-lo de diversas formas, entretanto, ele possui um conjunto de princípios de design. Esse conjunto, conforme Fielding (2000), o primeiro a citar o *Representational State Transfer*, em sua tese de doutorado, as define em seis restrições que governam a escalabilidade da *web* e consequentemente compõem o REST, sendo tais restrições (princípios):

- Cliente-Servidor (*Client-Server*): separação de responsabilidades entre cliente e servidor;
- Interface Uniforme: padronização da comunicação entre componentes;
- Sistema em Camadas (*Layered System*): permite a arquitetura em camadas;
- Cache: armazenamento de respostas para melhorar a eficiência;
- Sem estado (*Stateless*): cada requisição do cliente ao servidor deve conter todas as informações necessárias;

- Código sob demanda (*Code-on-Demand*): opcional e permite que o código seja transferido para a execução no cliente.

As API's REST utilizam os métodos do protocolo HTTP, como GET, PUT, POST e DELETE, para manipular recursos. Desta forma, permitindo a existência de um conjunto completo de operações CRUD (*create*, *read*, *update* e *delete*) e facilitando a criação de serviços web escaláveis e eficientes, com uma interface simples que é intuitiva para desenvolvedores.

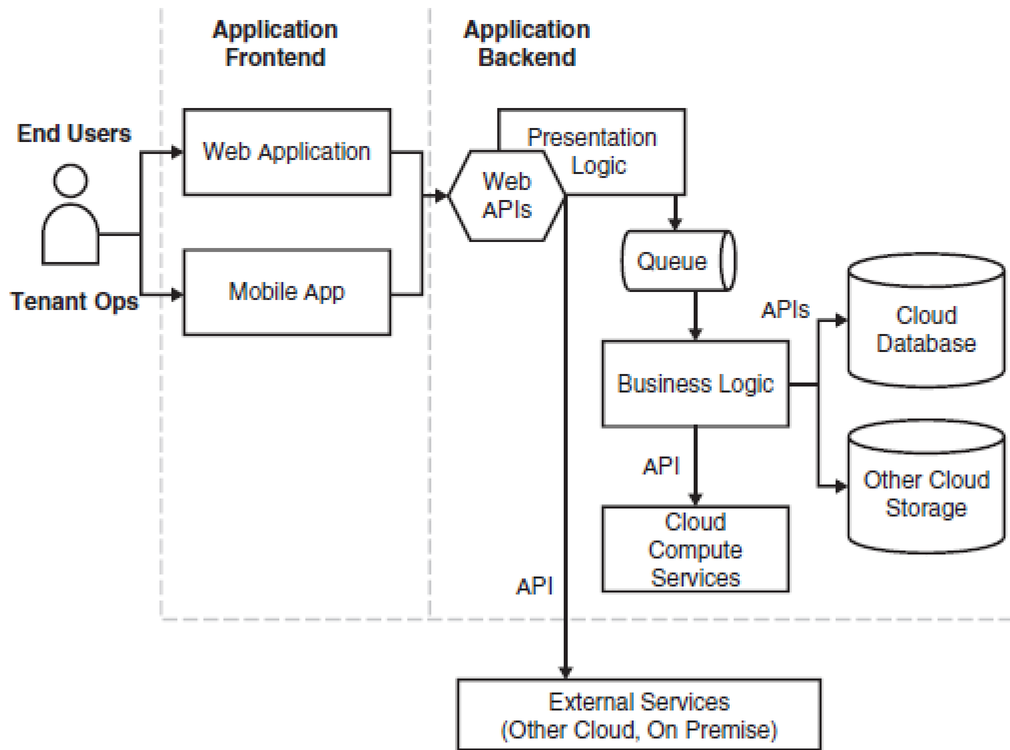
Portanto, o REST apresenta uma alternativa mais ágil e moderna em relação ao SOAP, atendendo às demandas de aplicações móveis e web. Com base nesse conjunto de operações, os serviços web RESTful tornam-se um padrão amplamente utilizado, conforme descrito por Massé (2012), consolidando-se como a principal escolha para o desenvolvimento de APIs na atualidade.

2.1.5 API

Uma API (Interface de Programação de Aplicações) é um conjunto de regras e protocolos que permitem a comunicação entre diferentes sistemas de software. Como descrito por Zimmermann et al. (2022), as APIs facilitam a integração de diferentes aplicações ao expor funcionalidades específicas de um sistema para que outros sistemas possam interagir com ele de maneira padronizada. APIs são fundamentais para a construção de sistemas modernos, permitindo que diferentes serviços e aplicações trabalhem juntos de forma eficaz, sem a necessidade de compartilhar todo o código ou a lógica interna de cada componente.

Na Figura 1 a seguir, observa-se a utilização de APIs conforme descrito acima. Aplicações web ou mobile conectam-se a uma API Web pública, que gerencia a lógica de apresentação, controle de filas e lógica de negócios. Essa API, por sua vez, utiliza outras APIs para se comunicar com bancos de dados ou serviços externos, exemplificando como as APIs atuam como mediadoras na integração de diferentes camadas e sistemas em um ambiente distribuído.

Figura 1 – Uso de APIs em uma arquitetura moderna baseada em nuvem.



Fonte: Zimmerman et al. (2022).

A governança de API refere-se ao conjunto de práticas, políticas e processos estabelecidos para gerenciar o ciclo de vida das APIs dentro de uma organização. Segundo Madden (2020), a governança é essencial para garantir que as APIs sejam seguras, eficientes e consistentes com os padrões corporativos. Isso inclui desde a definição de quem pode acessar a API até como as mudanças são implementadas e monitoradas ao longo do tempo, garantindo que as APIs suportem os objetivos de negócios de maneira sustentável.

A qualidade de uma API é avaliada com base em diversos critérios, como desempenho, segurança, facilidade de uso e manutenção. Madden (2020) argumenta que APIs de alta qualidade são aquelas que, além de funcionais, são robustas, seguras e capazes de evoluir com as necessidades do negócio sem comprometer a integridade do sistema. Além disso, uma API de qualidade deve ser bem documentada, facilitando o trabalho de desenvolvedores que precisam integrá-la a outros sistemas, garantindo uma experiência de desenvolvimento eficiente e sem frustrações.

2.2 Segurança no contexto de APIs

Segurança no contexto de APIs é fundamental para garantir a proteção dos dados e a continuidade dos serviços oferecidos por aplicações web. No cenário atual, em que a comunicação entre sistemas frequentemente ocorre através de APIs, é essencial assegurar que essas interfaces sejam robustas contra ataques maliciosos. Conforme Johnsson et al. (2019), a segurança vai além de apenas manter as informações em segredo; ela começa com a tríade clássica de confidencialidade, integridade e disponibilidade. Com o tempo, o conceito de rastreabilidade também foi adicionado a essa tríade como um pilar adicional da segurança da informação, refletindo a necessidade de auditar quando os dados foram acessados ou modificados. A seguir, exploraremos esses conceitos, identificação e autenticação, e os tipos de ataques que uma API pode sofrer em mais detalhes.

2.2.1 Confidencialidade

Quando falamos de confidencialidade no contexto das APIs, estamos nos referindo à necessidade de manter as informações em sigilo, garantindo que apenas usuários autorizados tenham acesso ao conteúdo. Como definido por Madden (2020), isso significa que apenas o público-alvo pretendido deve ser capaz de ler as informações.

Entretanto, ao abordar esse tópico, não lidamos apenas com desafios tecnológicos, como possíveis falhas de segurança que possam expor dados a ataques. É igualmente importante considerar o fator humano, como o compartilhamento indevido de informações ou a falta de proteção adequada em relação às senhas utilizadas. Um exemplo de ataque que pode comprometer a confidencialidade é o chamado ataque de '*man-in-the-middle*'. Nesse tipo de ataque, o invasor se posiciona entre o servidor e o usuário, interceptando as mensagens e roubando informações.

Para garantir a confidencialidade das informações, é fundamental utilizar técnicas como a criptografia dos dados durante a transmissão e o uso de canais

seguros, evitando redes públicas e conexões inseguras. Para minimizar falhas humanas, é essencial adotar autenticadores mais robustos, como a autenticação multifator (MFA). Esse sistema pode incluir, por exemplo, códigos de curta duração gerados por chaves e enviados via SMS para o número de telefone do usuário

2.2.2 Integridade

Quando falamos sobre a integridade dos dados em uma API, estamos nos referindo à garantia de que a informação é autêntica, confiável e precisa. Em outras palavras, integridade significa assegurar que os dados não foram alterados em nenhum momento sem autorização. Madden (2020) define integridade como a prevenção contra a criação, modificação ou destruição de informações sem a devida permissão para realizar tais operações.

As principais práticas para assegurar a integridade dos dados incluem o uso de assinaturas digitais, controle de versão, autenticação e controles de acesso. Além dessas medidas, a técnica de *hashing* desempenha um papel crucial na garantia da integridade. O *hashing* permite detectar qualquer alteração nas informações por meio da geração de um código de identificação único, conhecido como *hash*, no momento da criação da mensagem. Quando a mensagem é recebida, o mesmo algoritmo de *hash* é utilizado para verificar a integridade dos dados. Se houver qualquer modificação na mensagem, o algoritmo não conseguirá gerar o mesmo código, indicando uma violação da integridade dos dados.

2.2.3 Disponibilidade

Outro aspecto crucial é a disponibilidade do serviço, que garante que as informações estejam acessíveis e seguras para os usuários sempre que necessário. Este aspecto está diretamente relacionado ao foco deste trabalho, que é testar e avaliar práticas destinadas a prevenir ataques que possam comprometer a disponibilidade do serviço.

Disponibilidade pode ser definida como a garantia de que usuários legítimos de uma API possam acessá-la sem restrições, como dito por Madden (2020). Em

outras palavras, é essencial contar com uma infraestrutura robusta que suporte o aumento do número de usuários e implementações de segurança para prevenir ataques maliciosos. Exemplos dessas práticas incluem a aplicação de regras de *rate limit* para proteger contra ataques de negação de serviço distribuído (DDoS).

2.2.4 Rastreabilidade

Conforme Johnsson et al. (2019), o conceito de rastreabilidade foi posteriormente adicionado à tríade de segurança composta por confidencialidade, integridade e disponibilidade, devido à necessidade de saber quem acessou ou alterou determinados dados. Esse conceito tornou-se especialmente importante em setores como o financeiro e de saúde, onde a capacidade de rastrear acessos e modificações é crucial para manter a confiança e a conformidade com regulamentações. A rastreabilidade envolve a implementação de mecanismos de auditoria que registram detalhadamente as atividades realizadas sobre os dados, garantindo que qualquer ação possa ser monitorada ou revisada posteriormente.

Johnsson et al. (2019) também reforçam que a rastreabilidade ganhou maior destaque com a implementação de regulações como o Regulamento Geral de Proteção de Dados (GDPR) da União Europeia. Essa regulamentação, assim como a Lei Geral de Proteção de Dados (LGPD) no Brasil, determina que todo acesso a dados pessoais seja registrado em um sistema de auditoria. Dessa forma, a rastreabilidade não apenas protege os dados, mas também assegura a conformidade com normas legais, fortalecendo a segurança e a confiança dos usuários em sistemas críticos.

2.2.5 Identificação e autenticação

Para garantir a segurança de uma API, é fundamental abordar os conceitos de identificação e autenticação de usuários. Estes são dois elementos cruciais que devem ser bem implementados para assegurar a confiabilidade e integridade das informações em uma aplicação, além de contribuir para a disponibilidade, uma vez que a autenticação pode ajudar a prevenir ataques de negação de serviço. Embora

frequentemente confundidos, identificação e autenticação abordam aspectos diferentes dentro de uma aplicação. Em termos gerais, a identificação refere-se a saber 'quem' é o usuário, enquanto a autenticação verifica se essa identidade é realmente verdadeira (MADDEN, 2020).

A identificação do usuário é fundamental por várias razões. Primeiramente, é necessário personalizar a experiência do usuário, exibindo informações relevantes como nome e dados pessoais. Isso evita a exposição inadequada de informações de terceiros e garante que os dados apresentados sejam apropriados para cada usuário específico. Além disso, a identificação desempenha um papel crucial na segurança, pois é a base para o controle de acesso. Saber quem é o usuário permite determinar quais recursos e dados ele pode acessar, garantindo que cada usuário tenha acesso apenas às informações e funcionalidades para as quais está autorizado. Assim, a identificação não só permite personalizar a experiência, mas também protege a integridade e a confidencialidade dos dados, prevenindo acessos não autorizados.

Para a autenticação de usuários, existem diversas técnicas, que podem ser classificadas em três categorias principais, conforme descrito por Madden (2020). Essas categorias incluem:

- **Algo que o usuário sabe**, como uma senha pessoal;
- **Algo que o usuário possui**, como um dispositivo físico, por exemplo, uma chave de segurança ou um telefone celular;
- **Algo que o usuário é**, que se refere a características biométricas, como impressão digital ou reconhecimento facial.

No entanto, a utilização de apenas um fator de autenticação pode apresentar riscos. Senhas, por exemplo, podem ser comprometidas se forem fracas ou fáceis de adivinhar, enquanto fatores biométricos podem ter altas taxas de erro e resultar em falhas na autenticação caso o número de tentativas exceda os limites permitidos.

Para fortalecer o sistema de autenticação, a implementação de múltiplos fatores é altamente recomendada. Essa abordagem é conhecida como autenticação de dois fatores (2FA). Atualmente, a prática mais comum em aplicações web combina a identificação e autenticação inicial por meio de um identificador (como e-mail, ID ou nome) e senha. Após essa validação, um código adicional é enviado

por SMS ou e-mail para garantir que o usuário é quem afirma ser, proporcionando uma camada extra de segurança.

O processo de identificação de usuários pode ser complexo, especialmente se a aplicação precisa gerenciá-lo internamente.

> "Normalmente, estamos preocupados em identificar quem é esse usuário, mas, em muitos casos, a maneira mais fácil de fazer isso é fazer com que o cliente nos diga quem é e verificar se ele está dizendo a verdade." (MADDEN, 2020, p. 21).

Em outras palavras, não é necessário que a aplicação compreenda profundamente a identidade do usuário; em vez disso, podemos simplesmente permitir que o próprio usuário forneça sua identidade e, em seguida, validar essa identidade por meio dos métodos de autenticação apropriados.

2.2.6 Ameaças à APIs

Segundo Madden (2020), uma definição adequada de segurança para APIs deve considerar o ambiente em que a API opera e as potenciais ameaças que existirão nesse contexto. As ameaças representam qualquer maneira pela qual um dos princípios de segurança citados acima pode ser comprometido. Compreender o ambiente e os possíveis vetores de ataque é essencial para implementar medidas de proteção eficazes, garantindo que os dados e serviços oferecidos pela API permaneçam seguros contra acessos não autorizados e outras formas de exploração maliciosa.

As ameaças à segurança de APIs podem ser classificadas em várias categorias, um modelo amplamente conhecido é categorizado pelo acrônimo STRIDE, onde cada uma representando um tipo específico de risco. Conforme detalhado por Madden (2020), este acrônimo inclui *Spoofing*, onde um atacante se passa por outra pessoa; *Tampering*, que envolve a alteração de dados, mensagens ou configurações indevidamente; *Repudiation*, em que uma parte nega a realização de uma ação que de fato realizou; *Information disclosure*, que é a exposição de informações que deveriam permanecer privadas; *Denial of Service* (DoS), que visa impedir que outros acessem informações e serviços; e *Elevation of Privilege*, onde o

atacante ganha acesso a funcionalidades que não deveria ter. Cada uma dessas ameaças pode ser mitigada por mecanismos de segurança apropriados. Abaixo, traremos destaque para ataques de *Information Disclosure* e o *Denial of Service* (DDoS), tendo em vista que *rate limit* é uma ferramenta para prevenir-se deles.

2.2.6.1 Força Bruta

Ataques de força bruta são uma ameaça comum às APIs, onde um invasor tenta adivinhar credenciais de autenticação, como senhas, através de tentativas repetitivas e automatizadas. Madden (2020) descreve que, nesse tipo de ataque, o invasor utiliza scripts automatizados para tentar todas as combinações possíveis de credenciais até conseguir acesso ao sistema. Esse tipo de ataque pode ser particularmente devastador quando a API não implementa medidas adequadas de proteção, como bloqueio de IP após várias tentativas falhas, o que torna a defesa contra ataques de força bruta uma prioridade na segurança de APIs.

2.2.6.2 DDoS

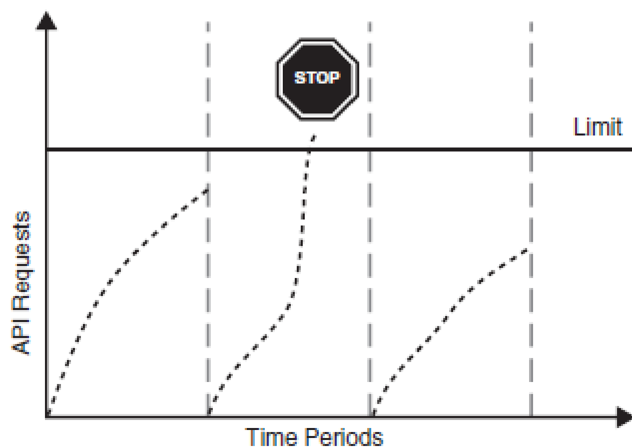
Os ataques de negação de serviço distribuído (DDoS) são outra ameaça significativa para a estabilidade de APIs. Durante um ataque DDoS, o invasor utiliza uma rede de dispositivos comprometidos para sobrecarregar a API com um grande volume de solicitações simultâneas, com o objetivo de esgotar os recursos do servidor e tornar o serviço indisponível para usuários legítimos. Zimmermann et al. (2022) enfatizam que a aplicação de políticas de *rate limit* é uma das estratégias mais eficazes para mitigar os efeitos de ataques DDoS, limitando a quantidade de solicitações que um cliente pode fazer em um intervalo de tempo específico, protegendo assim a integridade e a disponibilidade da API.

2.3 Rate Limit

O rate limit é uma técnica de controle de acesso que restringe o número de solicitações que um usuário pode fazer a uma API dentro de um determinado

período de tempo. Segundo Madden (2020), o objetivo principal desta técnica é prevenir que um único usuário sobrecarregue o sistema, garantindo que os recursos da API sejam distribuídos de maneira justa entre todos os usuários. Esse mecanismo é essencial para manter a estabilidade e a disponibilidade de serviços em ambientes onde múltiplos clientes competem pelos mesmos recursos. Abaixo, na Figura 2, é exemplificado um cenário em que um usuário excede o limite de taxa configurado.

Figura 2 – Rate Limit: Quando o cliente excede o número permitido de pedidos por período de tempo, todos os outros pedidos são recusados



Fonte: Zimmermann et al. (2023, p. 413).

Para melhor compreender o *rate limit* é necessário compreender Acordos de Nível de Serviço (SLA), planos de precificação, diferentes tipos de abordagens para limitação de taxas de requisições e os diferentes algoritmos, além das vantagens e desvantagens de cada um deles.

2.3.1 Acordos de nível de serviço

Um Acordo de Nível de Serviço (SLA) é um documento que formaliza os compromissos entre um provedor de serviço e seus clientes, definindo os parâmetros de desempenho que o serviço deve atender. Um SLA inclui pelo menos um Objetivo de Nível de Serviço (SLO), que especifica metas claras, como tempo de

resposta ou disponibilidade, que o serviço deve alcançar. Além disso, o SLA estabelece penalidades, créditos compensatórios ou ações que serão aplicadas caso essas metas não sejam cumpridas, bem como os procedimentos de relato de violações (BEYER et al., 2016).

A aplicação de *rate limit* é fundamental para a definição de Acordos de Nível de Serviço (SLA) e planos de precificação (*pricing plans*) em APIs. Zimmermann et al. (2022) discutem que, ao limitar o número de solicitações permitidas em um período específico, as empresas podem garantir a qualidade do serviço prometido em seus SLAs.

2.3.2 Nível de Objetivo de Serviço

Service Level Objective (SLO), ou em português, Nível de Objetivo de Serviço, é um alvo ou uma faixa-alvo estabelecida para medir o desempenho de um serviço. Por exemplo, em uma pesquisa de banco de dados, pode-se definir que a latência média, ou tempo de resposta, deve ser de 500 milissegundos ou menos. Ele serve como um parâmetro claro para avaliar se o serviço atende aos requisitos esperados e é fundamental para alinhar expectativas entre os provedores de serviços e seus usuários.

Segundo Beyer et al. (2016), a ausência de uma definição explícita de SLO pode levar os usuários a desenvolverem suas próprias expectativas sobre a performance desejada, muitas vezes desalinhadas com o que foi planejado e implementado pelo time responsável. Isso pode resultar em superestimação ou excesso de confiança no serviço oferecido, gerando frustração quando o desempenho não corresponde às expectativas dos usuários. Assim, o SLO não apenas define limites operacionais claros, mas também serve como uma ferramenta de comunicação para gerenciar expectativas de forma eficaz.

A definição de um SLO adequado, no entanto, é um processo complexo. Muitas vezes, os valores não são escolhidos arbitrariamente, mas determinados por uma combinação de necessidades dos usuários, capacidades da arquitetura existente e restrições técnicas. Quando a definição é muito rigorosa pode ser inalcançável, colocando pressão desnecessária sobre a infraestrutura, enquanto

uma meta muito permissiva pode comprometer a experiência do usuário. Por isso, encontrar o equilíbrio certo exige uma análise criteriosa de métricas históricas, testes de carga e feedback contínuo dos usuários, garantindo que os objetivos estabelecidos sejam tanto realistas quanto úteis.

2.3.3 Plano de precificação

Para APIs, plano de precificação, é uma possível estratégia utilizada por organizações para monetizar o uso de suas APIs, considerando o valor tanto monetário quanto imaterial que essas interfaces representam. Esse plano define como os clientes da API serão cobrados pelo seu uso. O plano de precificação deve ser estruturado de forma a equilibrar a geração de receita com a acessibilidade para os desenvolvedores, permitindo que a API seja financeiramente sustentável enquanto atende às necessidades dos usuários. Além disso, o *rate limit* permite a criação de diferentes planos de precificação, onde usuários que necessitam de maior acesso aos recursos podem optar por planos que ofereçam limites de requisição mais altos, gerando assim uma fonte de receita adicional (ZIMMERMANN et al., 2022).

2.3.4 Abordagens de rate limit

Existem vários tipos de algoritmos de *rate limit* que podem ser aplicados dependendo das necessidades específicas de uma API. Zimmermann et al. (2022) destacam três principais abordagens:

- Limitadores baseados em IP, onde as solicitações são limitadas com base no endereço IP do cliente;
- Limitadores baseados em usuário, que impõem limites com base na identidade do usuário;
- Limitadores baseados em *tokens*, onde *tokens* temporários são usados para controlar o número de acessos permitidos.

Cada tipo oferece vantagens diferentes, dependendo da natureza do serviço e do comportamento esperado dos usuários.

2.3.5 Algoritmos de rate limit

Os algoritmos de *rate limit* desempenham um papel crucial na implementação eficaz dessa técnica. Baseado na explicação de Patidar (2024), dentre os mais comuns temos:

- Janela fixa: divide o tempo em janelas fixas (por exemplo, 1 minuto). Se um usuário realizar requisições acima do limite permitido dentro da janela, ele será bloqueado até o início da próxima janela;
- Janela deslizante: similar à janela fixa, entretanto em vez de usar janelas fixas, a janela se move à medida que o tempo passa. Isso permite uma taxa de requisições mais uniforme;
- Balde de tokens: permite o tráfego de requisições enquanto existir tokens disponíveis no “balde”, se o balde estiver vazio, requisições devem aguardar que novos tokens sejam adicionados ao balde. Durante períodos de pouco uso, tokens são acumulados, permitindo ocasionalmente rajadas de requisições;
- Balde furado: similar ao balde de tokens, porém ele não permite acúmulo de tokens além do limite do balde, portanto, pode ser útil a suavizar rajadas de requisições;
- Limite de requisições simultâneas: limita as requisições a um número de requisições concorrentes, ou seja, número máximo de requisições executadas em paralelo. Quando o tempo de resposta de uma API não for uniforme, o número de requisições por tempo pode variar bastante.

Esses algoritmos são fundamentais para garantir que as APIs permaneçam disponíveis e estáveis, mesmo sob condições adversas, ao prevenir sobrecargas de tráfego e abusos.

2.4 Testes

Testes de software é um processo fundamental para garantir que o código fonte execute corretamente suas funções de forma prevista e, ao mesmo tempo, não realize nenhuma operação inesperada. Como afirmado por Myers et al. (2011), o

objetivo dos testes é assegurar que o software seja previsível e consistente, evitando surpresas para os usuários. Assim, testar software é essencial para identificar e corrigir erros, garantindo que o produto final atenda às expectativas e requisitos estabelecidos.

Existem duas formas principais de testar software, de forma manual com testes exploratórios ou com testes automatizados. De acordo com Myers et al. (2011) testes automatizados desempenham um papel crucial no desenvolvimento ágil de software. A automação deles é preferível sobre realização manual, não somente pela agilidade que eles agregam, mas também pelo aumento da confiabilidade e a redução de riscos associados à introdução de novos bugs durante o processo de entrega de software.

Embora existam diversas ferramentas de testes disponíveis, é crucial que a ferramenta selecionada seja utilizada de forma consistente pelos desenvolvedores e testadores. Deve-se construir uma cultura de qualidade que esteja preocupada em entregar software com qualidade apesar de desafios com prazos curtos ou outros desafios. Portanto, mesmo que alguns cenários exijam testes exploratórios manuais, no longo prazo a automação é a abordagem preferida para garantir qualidade e aumentar a eficiência no desenvolvimento de software (MYERS et al., 2011).

Há diversos tipos de testes de softwares, unitários, de integração, de carga, de estresse, entre outros. Segundo Pressman (2011), testes unitários visam validar o comportamento na menor unidade de um projeto de software, assim a complexidade relativa dos testes e dos erros que eles revelam são limitados pelo escopo restrito. Por outro lado, testes de integração, buscam validar a comunicação entre essas unidades, garantindo que nenhuma informação seja perdida ou ocorra algum efeito inesperado na integração ou comunicação destes componentes.

Testes de carga busca determinar como uma aplicação e seu ambiente responde em diferentes condições de cargas, ou seja, dependendo de uma série de parâmetros, como o número de usuários concorrentes (N), o número de transações por usuário por unidade de tempo (T) e a quantidade de dados processados pelo servidor por transação (D). Parâmetros fundamentais para simular condições reais de uso e garantir que os testes sejam eficientes sob diferentes cargas. Ainda neste sentido, testes de estresse buscam exceder os limites operacionais do software,

com a finalidade de identificar como ocorre a degradação do sistema, se é gradual ou instantânea, quais mensagens de erros são registradas, como é a experiência do usuário neste cenário, se a integridade dos dados é mantida, etc (PRESSMAN, 2011).

Para o presente trabalho, o objetivo é a aplicação de testes de carga, que possibilita avaliar o comportamento da API sob volumes variados de requisições, desde baixos a altos níveis de demanda. Dessa forma, os mesmos volumes aplicados a API com e sem a utilização da técnica de *rate limit* permite comparar os comportamentos e a influência desta técnica.

3 TRABALHOS RELACIONADOS

Neste capítulo, são descritos os trabalhos que empregam técnicas e/ou soluções relacionadas a APIs e *rate limit*. Os trabalhos selecionados resultam de pesquisas realizadas diretamente em bibliotecas digitais de universidades e no Google Acadêmico, utilizando critérios que priorizam estudos cujo tema esteja diretamente relacionado a APIs ou a técnica de *rate limit*. A seleção considera a atualidade e relevância desses trabalhos para o tema, focando naqueles que apresentam soluções práticas, indicam frameworks ou discutem padrões aplicáveis ao desenvolvimento e à governança de APIs e *rate limit*. Esses trabalhos são relevantes por contribuírem para o aprofundamento do tema, explorando ferramentas, métodos consolidados e abordagens inovadoras.

Primeiramente, é apresentada uma breve descrição de cada trabalho, seguida de uma comparação com o presente estudo e a solução proposta, com o objetivo de destacar as principais semelhanças e diferenças entre eles.

3.1 DataCache: Gestão e padronização na comunicação entre aplicações

O trabalho de Majolo (2019) aborda a melhoria da qualidade de web services através da utilização de cache para otimizar a comunicação entre aplicações, reduzindo a carga nas APIs e melhorando o tempo de resposta. Comparado ao presente estudo, enquanto Majolo foca na redução da carga utilizando cache, este trabalho se concentra em proteger as APIs contra abusos e sobrecargas por meio de

técnicas de *rate limit*. Embora ambos os estudos busquem melhorar a performance e a estabilidade das APIs, eles o fazem através de abordagens distintas.

3.2 Disponibilização de dados de fundos de investimento através de uma API

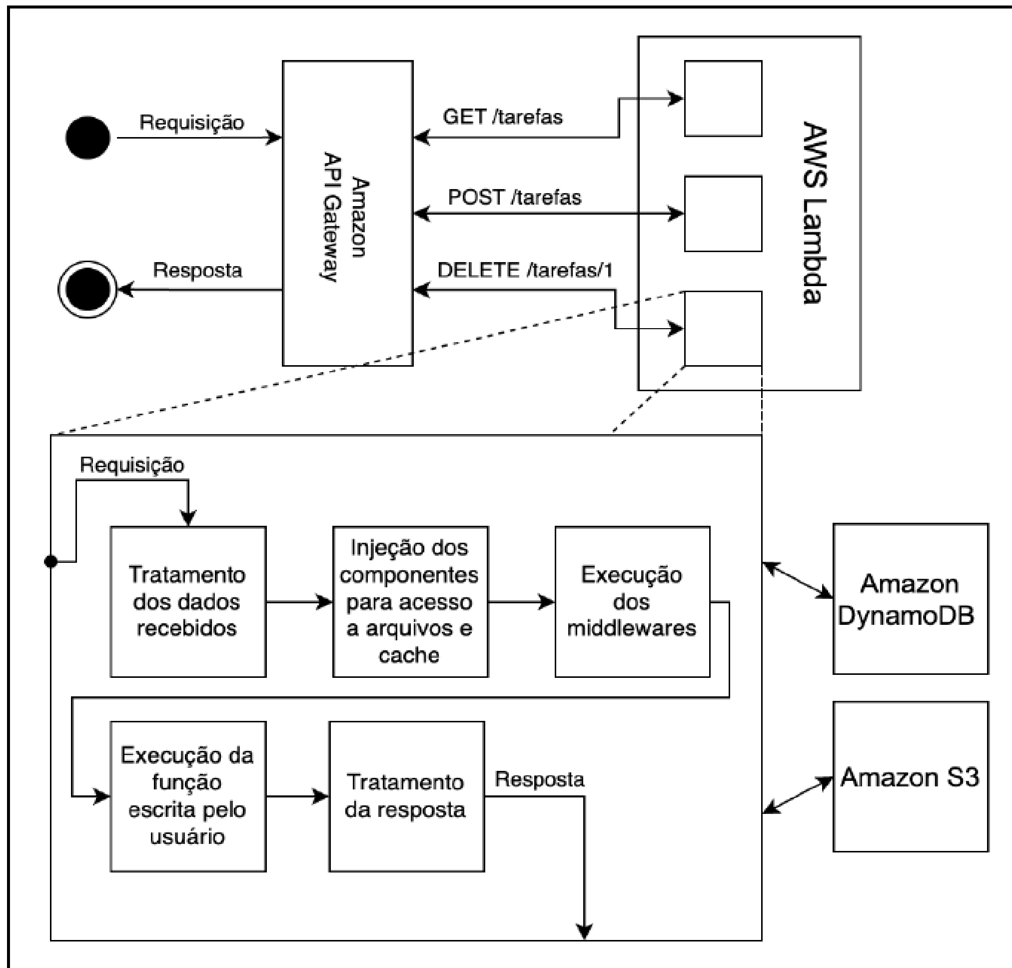
O trabalho de Mileto (2021) focou na criação de uma API para disponibilizar dados de fundos de investimento, com um foco especial em garantir performance e realizar testes de desempenho. Embora o trabalho compartilhe a preocupação com a performance de APIs, ele não aborda o *rate limit* como técnica de governança para prevenir abusos e ataques, que é o foco deste estudo. Assim, o presente trabalho complementa o estudo de Mileto ao propor uma solução para proteger as APIs contra acessos excessivos e não autorizados.

3.3 Framework para desenvolvimento e implantação de aplicações HTTP utilizando serverless computing

O trabalho de Ströher (2023) explora a utilização de arquitetura serverless para o desenvolvimento de aplicações HTTP, destacando os benefícios em termos de escalabilidade e custo. Na Figura 3, apresentada abaixo, ilustra o framework proposto pelo autor para a construção de aplicações HTTP, detalhando o fluxo de requisições e respostas dentro da arquitetura.

Embora compartilhe o objetivo de melhorar a performance e a confiabilidade das APIs, ele adota uma abordagem diferente ao utilizar a arquitetura serverless em vez de técnicas de governança de API como o *rate limit*. Este trabalho se diferencia ao abordar especificamente a proteção e estabilidade das APIs através do controle de acesso e limitação de requisições e pode complementar o trabalho de Ströher.

Figura 3 – Fluxograma da arquitetura do framework de Ströher.



Fonte: Ströher (2023, p. 46).

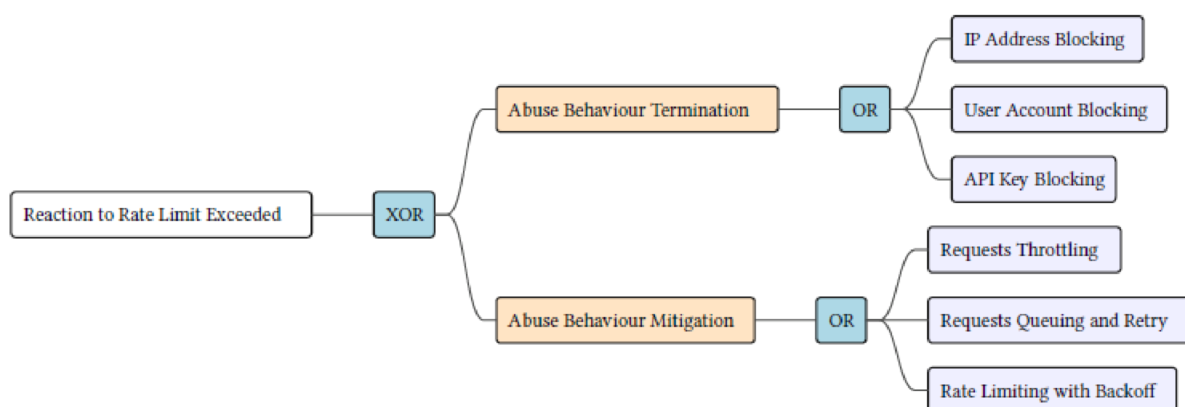
3.4 Impact of API Rate Limit on Reliability of Microservices-Based Architectures

O estudo de Malki et al. (2022) explora o impacto do *rate limit* na confiabilidade de arquiteturas baseadas em microsserviços. O trabalho apresenta um modelo analítico para prever os impactos do *rate limit* na confiabilidade dos serviços, validando-o por meio de experimentos em nuvens privadas e no Google Cloud Platform. Comparado ao presente estudo, ambos compartilham o foco no *rate limit* como uma técnica para melhorar a estabilidade e a confiabilidade das APIs. No entanto, enquanto Malki et al. se aprofunda na modelagem e na previsão dos efeitos do *rate limit*, este trabalho busca implementar e testar o *rate limit* em cenários reais, explorando diferentes algoritmos e suas aplicações práticas.

3.5 API Rate Limit Adoption – A pattern collection

O estudo de Serbout et al. (2023) apresenta uma coleção de padrões de design para adoção de *rate limit* em APIs. O trabalho destaca as melhores práticas para implementar *rate limit*, oferecendo um guia para desenvolvedores e arquitetos de software. A Figura 4 apresenta um exemplo de padrão discutido pelos autores, ilustrando as possíveis reações de uma API quando é ultrapassado o limite de taxa de requisições configurado. Entre essas reações, destacam-se estratégias de mitigação, como limitação das taxas de requisições, e ações mais restritivas, como bloqueio por IP ou pela conta de usuário.

Figura 4 – Reações de APIs ao exceder o limite de taxa de requisições.



Fonte: Serbout et al. (2023, p. 11).

Comparado ao presente trabalho, que foca na implementação prática e nos testes de carga para validar a eficácia do *rate limit*, o estudo de Serbout et al. se concentra mais em fornecer um conjunto de padrões e diretrizes teóricas. Ambos os estudos se complementam, com o presente trabalho fornecendo uma validação prática dos conceitos e técnicas discutidas por Serbout et al.

3.6 Comparativo entre os trabalhos relacionados

No Quadro 1 abaixo, é realizado um comparativo entre as propostas apresentadas nos trabalhos relacionados. O presente trabalho se baseia em conceitos de *rate limit* aplicados à governança de APIs, enquanto os trabalhos relacionados abordam outras técnicas, como cache, *serverless computing*, e modelagem analítica. Uma semelhança importante é o foco em garantir a performance e a estabilidade das APIs, mas o diferencial deste trabalho está na aplicação prática e na validação de algoritmos de *rate limit* através de testes de carga. Tendo em vista que, não se encontram trabalhos que combinem todas essas abordagens em um estudo único e integrado, o que destaca a relevância e a originalidade desta pesquisa.

Quadro 1 – Comparativo dos trabalhos relacionados

Trabalho	**Técnica principal**	**Foco**	**Relacionamento com este estudo**
DataCache: Gestão e padronização na comunicação entre aplicações	Cache	Otimização de web services	Compartilham o aprimoramento de web services, com técnicas diferentes.
Disponibilização de Dados	API	Desempenho de APIs	Compartilha o interesse em performance, porém não aborda o *rate limit.*
Serverless Framework	*Serverless Computing*	Escalabilidade e custo	Aborda escalabilidade de webservices, mas com foco em serverless, não em *rate limit*.
API Rate Limit Adoption -- A pattern collection	Padrões de design	Boas práticas para *rate limit*	Complementa este trabalho fornecendo diretrizes teóricas.
Impact of API Rate Limit on Reliability of Microservices-Based Architectures	*Rate Limit*	Confiabilidade do *rate limit* em Micro serviços	Compartilha o foco em *rate limit*, mas foca na modelagem e previsão de impactos.

Fonte: Do autor (2024).

Além disso, o Quadro 1 evidencia como este trabalho se posiciona em relação aos demais estudos, não apenas ao explorar a aplicação do *rate limit*, mas também ao propor um ambiente controlado para validação prática dos conceitos. A integração entre os testes de carga e a análise de impacto no desempenho das APIs representa um diferencial metodológico. Essa abordagem prática permite identificar lacunas nas estratégias existentes, especialmente em cenários de alta demanda, onde o equilíbrio entre estabilidade e eficiência é crucial. Assim, este trabalho complementa a literatura existente, fornecendo percepções úteis para a implementação de soluções em ambientes reais e contribuindo para o avanço das boas práticas em governança de APIs.

4 MATERIAIS E MÉTODOS

Os procedimentos metodológicos definem o modo pelo qual o acadêmico deseja trabalhar a investigação e exposição da pesquisa. Desta forma, é definido os procedimentos, métodos e caminhos a serem seguidos no decorrer do trabalho.

4.1 Pesquisa quanto aos Métodos Científicos

Neste trabalho será empregado o modo de pesquisa dedutivo, conforme Panasiewicz e Baptista (2013), o método dedutivo parte das generalizações, leis gerais, e a partir delas tentar confirmá-las em uma experiência em particular.

O presente trabalho analisará como funciona uma API e as melhores práticas e soluções de governança dessas interfaces, adaptando essas soluções conforme as necessidades do aplicativo proposto e desenvolvendo na prática um software que implemente os conceitos pesquisados.

4.1.1 Pesquisa quanto ao Modo de abordagem

O modo de abordagem adotado nesta pesquisa será de maneira qualitativa. Conforme Panasiewicz e Baptista (2013), a abordagem qualitativa é mais subjetiva do que objetiva. Nela pode-se usar dados quantitativos, entretanto, foca na interpretação e compreensão dos fenômenos e objetos estudados.

A pesquisa se encaixa no âmbito qualitativo, visto que a solução tecnológica proposta é remetida para validação e sua qualidade é interpretada através de métricas determinadas.

4.1.2 Pesquisa quanto aos Fins da Pesquisa

As metodologias de pesquisa utilizadas para o andamento do presente estudo empregam o caráter exploratório e descritivo.

Segundo Panasiewicz e Baptista (2013), o objetivo exploratório pode ser dividido em dois tipos: teórico ou prático. Em geral, é uma aproximação do tema. Na teoria pode ser uma bibliografia, enquanto na prática pode ser entrevistas, levantamentos de dados, informações ou opiniões.

Quanto a pesquisa descritiva, ainda conforme Panasiewicz e Baptista (2013), ela é fundamentalmente objetiva e sistemática. Dessa forma, busca-se observar, descrever, coletar dados, classificar e analisar o fenômeno.

A pesquisa exploratória está relacionada com os objetivos de compreender o uso de APIs, os aspectos de governança de APIs e o padrão de *rate limit*, avaliando como essas práticas podem melhorar a estabilidade, disponibilidade e eficiência de sistemas em ambientes de alta demanda.

Enquanto a pesquisa descritiva está relacionada com as informações que serão obtidas e documentadas durante o desenvolvimento do presente trabalho.

4.1.3 Pesquisa enquanto aos Procedimentos técnicos

Esta pesquisa seguirá a abordagem do tipo estudo de caso. Para Gil (2008), a pesquisa de estudo de caso é um estudo realizado com a finalidade de obter-se um conhecimento amplo e detalhado de um ou poucos objetos de estudo. Portanto, deve ser feito um estudo profundo e exaustivo do tema.

O presente estudo vai ao encontro a este tipo de abordagem, a pesquisa de estudo de caso permite permite investigar de forma prática e detalhada o uso de *rate limit* como técnica de governança em APIs, explorando sua aplicação em um contexto realista. A abordagem possibilita não apenas compreender os conceitos teóricos associados ao tema, mas também avaliar suas implicações práticas em cenários de alta demanda e abuso de recursos.

4.2 Tecnologias

Nesta seção, são abordadas as tecnologias aplicadas no desenvolvimento deste trabalho, com o objetivo de atingir o resultado final proposto. São apresentadas as ferramentas que foram utilizadas, bem como as linguagens de programação e os softwares que foram empregados na implementação dos algoritmos.

As tecnologias selecionadas desempenham papéis fundamentais na construção e validação do trabalho. O NPM e o TypeScript são utilizados para estruturar o repositório, estabelecendo a base para o desenvolvimento do projeto. O NestJS, combinado com o Sequelize, é empregado para construir a API e seus módulos, incluindo a integração com banco de dados, que é abstraída por meio do Sequelize. O Docker e o PostgreSQL são aplicados para disponibilizar um banco de dados isolado, replicando um ambiente realista para os testes. Por fim, o Postman e o Grafana k6 são utilizados para validar a solução, enviando requisições HTTP e executando scripts de testes de carga, garantindo que a implementação seja avaliada sob condições reais e intensivas.

4.2.1 TypeScript

Segundo Cherny (2019), TypeScript, uma linguagem de programação desenvolvida pela Microsoft, surgiu como uma extensão do JavaScript, criada para abordar algumas das limitações que desenvolvedores enfrentam ao trabalhar em projetos grandes e complexos. A principal característica do TypeScript é a adição de tipagem estática opcional ao JavaScript, permitindo que os desenvolvedores definam explicitamente os tipos de variáveis, funções e objetos, o que não é possível no JavaScript tradicional. Esta tipagem estática ajuda a detectar e prevenir erros comuns em tempo de compilação, que no JavaScript só seriam detectados em tempo de execução, ou seja, torna o código fonte mais robusto e seguro.

Cherny (2019) destaca as vantagens do TypeScript, como a capacidade de melhorar a manutenção de código e a escalabilidade de aplicações. Ao utilizar TypeScript, os desenvolvedores podem aproveitar recursos como autocompletar e

refatoração segura, tornando o desenvolvimento mais eficiente e menos propenso a erros. Além disso, por ser um superconjunto de JavaScript, TypeScript é totalmente compatível com o vasto ecossistema de bibliotecas e *frameworks* JavaScript existentes.

4.2.2 Node Package Manager (NPM)

O Node Package Manager (NPM), de acordo com a documentação oficial do NPM (2024), é um gerenciador de pacotes padrão para o ambiente Node.js, e é amplamente utilizado por desenvolvedores para instalar, compartilhar e gerenciar dependências em projetos JavaScript. Assim, facilita o acesso a uma vasta coleção de pacotes disponíveis em seu repositório, permitindo que você incorpore rapidamente bibliotecas e ferramentas essenciais ao seu projeto.

Ao criar uma API, o NPM ajuda a integrar funcionalidades prontas, como *frameworks*, *middleware* e outras bibliotecas que podem simplificar e acelerar o processo de desenvolvimento. Além disso, ele gerencia automaticamente as versões e as dependências, garantindo que todos os pacotes necessários para o seu projeto estejam corretamente instalados e atualizados, o que contribui para a manutenção e a escalabilidade da sua API.

4.2.3 NestJS

NestJS é um *framework* progressivo para Node.js, criado para desenvolver aplicações *server-side* eficientes, escaláveis e facilmente mantidas. Baseado em TypeScript, o *framework* adota conceitos modernos de desenvolvimento, como programação orientada a objetos, programação funcional e arquiteturas modulares, facilitando a criação de APIs robustas e serviços de *backend*. Ele abstrai grande parte da complexidade associada à configuração e estruturação de aplicações em Node.js, permitindo que os desenvolvedores se concentrem em construir funcionalidades e garantir a qualidade do código, sem se preocupar com os detalhes técnicos subjacentes (CORREA; LIM, 2022).

A arquitetura modular do NestJS facilita a organização do código, promovendo a reutilização de componentes e a manutenção a longo prazo. Uma de suas principais funcionalidades são os *guards*, que determinam se uma requisição será processada pelo route handler, dependendo de condições específicas verificadas em tempo de execução. Eles permitem a interposição de lógica de processamento no ciclo de requisições de forma declarativa, contribuindo para manter o código mais limpo e organizado (NESTJS, 2024).

Além disso, o NestJS oferece integração nativa com outras bibliotecas e ferramentas do ecossistema Node.js, tornando-o uma escolha popular para o desenvolvimento de aplicações corporativas e APIs que exigem alta performance e escalabilidade. Sua flexibilidade e robustez tornam-no uma solução eficiente para arquiteturas modernas e demandas crescentes em sistemas distribuídos.

4.2.4 Grafana k6

O Grafana k6, comumente referido apenas como k6, é uma ferramenta de código aberto para realizar testes de carga, projetada para ajudar desenvolvedores e engenheiros a testar a performance de suas aplicações, especialmente em ambientes com alto volume de requisições HTTP. Desenvolvido pela Grafana Labs, o k6 permite que você simule uma grande quantidade de usuários acessando a sua API ou aplicação web ao mesmo tempo, medindo como o sistema responde sob diferentes níveis de carga e estresse (GRAFANA LABS, 2024).

Ele também permite a criação de scripts de teste em JavaScript, definindo cenários de carga que replicam o comportamento de usuários reais. Isso é especialmente útil para identificar gargalos de performance, testar a escalabilidade da aplicação, e garantir que sua API possa lidar com um alto número de requisições simultâneas sem comprometer a estabilidade ou a disponibilidade do serviço. Além disso, ele oferece relatórios detalhados e métricas em tempo real, permitindo uma análise precisa do desempenho e ajudando a otimizar o sistema para melhor responder às demandas de tráfego intenso.

4.2.5 Docker

De acordo com a documentação do Docker (2024), trata-se de uma plataforma para desenvolver, enviar e executar aplicações. Ele permite a separação entre a aplicação e a infraestrutura, o que auxilia na entrega e teste de software de forma mais ágil. O Docker possibilita o gerenciamento de recursos de infraestrutura, como a definição da quantidade de CPUs, memória alocada, entre outros, tudo de maneira isolada e segura. Esse ambiente isolado, onde aplicações ou serviços são enviados, instalados e executados, é chamado de *container*. Além disso, o Docker é portátil, podendo ser instalado e utilizado localmente nos computadores dos desenvolvedores, em máquinas físicas ou virtuais de *data centers*, ou em provedores de nuvem. As principais características do Docker incluem:

- Imagens: São templates baseados, geralmente, em sistemas operacionais Linux, que podem conter instalações pré-configuradas de servidores de bancos de dados, como PostgreSQL, MySQL, MongoDB, servidores web como Apache, entre outros. Essas imagens podem ser customizadas e salvas como novos templates, facilitando a criação e replicação de ambientes de desenvolvimento;
- *Containers*: Representam instâncias executáveis de uma imagem. É possível criar, iniciar, parar e excluir containers utilizando a API ou CLI do Docker, permitindo uma gestão prática e eficiente dos recursos computacionais.

Essa estrutura robusta e flexível torna o Docker uma tecnologia valiosa para o desenvolvimento e a operação de aplicações.

4.2.6 Sequelize

O Sequelize é uma biblioteca para Node.js e TypeScript que funciona como um ORM (*Object-Relational Mapper*), facilitando a integração com bancos de dados relacionais, como PostgreSQL, MySQL, MariaDB, SQL Server, entre outros. Ele atua como um driver para a conexão com esses bancos, abstraindo a utilização de comandos SQL por meio de técnicas de mapeamento objeto-relacional. Essa abordagem permite que o desenvolvedor interaja com o banco de dados utilizando

objetos da aplicação em vez de comandos SQL diretamente, mapeando tabelas para classes, colunas para atributos e registros para instâncias dessas classes (DURANTE, 2022).

Essa abstração simplifica e otimiza o desenvolvimento de aplicações, reduzindo a quantidade de código necessária para operações no banco de dados e promovendo uma linguagem mais uniforme e consistente na aplicação. Além disso, o Sequelize oferece suporte a diversas funcionalidades, como migrações de banco, validações, associações entre tabelas e transações, o que o torna uma ferramenta robusta para lidar com bancos de dados relacionais de maneira eficiente e programática.

4.2.7 PostgreSQL

PostgreSQL é um sistema de gerenciamento de banco de dados (SGBD) relacional de código aberto, portanto, sua licença é liberal e pode ser utilizado, modificado e distribuído por qualquer pessoa gratuitamente. Segundo a documentação do PostgreSQL (2024), o sistema foi desenvolvido no departamento de informática da Universidade da Califórnia em Berkeley. Sendo pioneiro em muitos conceitos que só se tornaram disponíveis em outros sistemas de banco de dados comerciais muito mais tarde.

Conforme Elmasri e Navathe (2018), bancos de dados e sistemas de bancos de dados são um componente essencial na vida da sociedade moderna, a maioria de nós encontra diariamente atividades que envolvem interação com bancos de dados. Ainda conforme os autores, SGBDs, são sistemas de software de uso geral que facilitam o processo de definição, construção, manipulação e compartilhamento de dados entre diversos usuários e aplicações.

Desta forma, é bem comum APIs se comunicarem com um ou mais banco de dados para manipular os dados que serão retornados para outras aplicações, ou seja, bancos de dados são um recurso normalmente integrado a APIs e crucial para o funcionamento esperado das aplicações. Assim, sendo útil para este projeto que visa simular um sistema real para realizar testes de carga em interfaces com e sem *rate limit*.

4.2.8 Postman

O Postman é uma ferramenta de software amplamente utilizada para testar e interagir com APIs por meio de uma interface prática e intuitiva. Ele permite o envio de requisições, exibe respostas em diversos formatos e possibilita a criação e organização de coleções compartilháveis. Segundo Hamilton (2024), o Postman também suporta variáveis, ambientes, scripts e testes, permitindo a personalização e automação de processos, o que o torna uma solução versátil e indispensável para desenvolvedores e equipes que trabalham com APIs, ao facilitar a validação dos serviços reduzindo ações repetitivas e economizando tempo.

4.3 Desenvolvimento

Com o objetivo de desenvolver e validar uma solução de *rate limit* para APIs, foi projetada e implementada uma API que atende a um conjunto específico de casos de uso, como autenticação de usuários, manipulação de registros e geração de sumários de dados. Este capítulo detalha as principais etapas do desenvolvimento, abordando inicialmente o escopo funcional da API, que é ilustrado por um diagrama de caso de uso, descrevendo as funcionalidades e interações dos usuários com o sistema.

Além disso, apresenta-se a arquitetura completa da solução, destacando os principais componentes, como o servidor REST desenvolvido com o *framework* NestJS, o controle de acesso baseado em JWT, a implementação de limite de taxas de requisições para garantir estabilidade e segurança, e a integração com o banco de dados PostgreSQL. Também é detalhado o processo de desenvolvimento da solução de *rate limit*, que utiliza a técnica de Janela Fixa para proteger rotas críticas contra abusos e uso excessivo.

Por fim, o capítulo aborda a criação de um ambiente de banco de dados em *container* utilizando Docker, configurado com PostgreSQL, e a geração de dados iniciais por meio de scripts "*seeders*". Esses dados foram criados para viabilizar e garantir a consistência dos testes de desempenho e validação da solução,

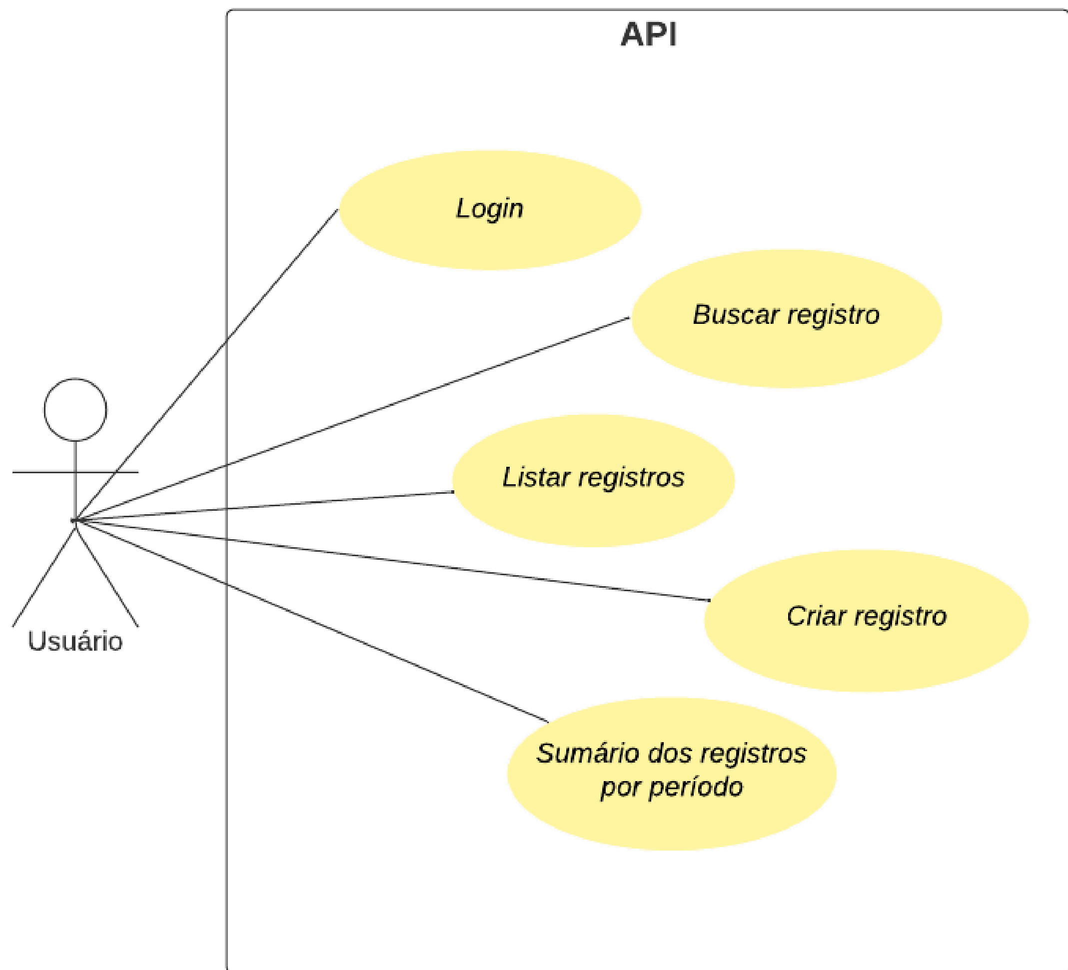
proporcionando uma base sólida para as etapas subsequentes de testes e análise dos resultados.

4.3.1 Escopo

Para a validação da solução de *rate limit*, foi necessário criar um cenário que simulasse um ambiente real de uso de APIs. Para isso, foi desenvolvido um módulo que permite a execução de operações comuns em APIs, como criação de registros, listagem de registros individuais ou em lote e a sumarização dos dados registrados, este módulo é chamado de transações, pois os registros assemelham-se a transações financeiras em uma conta bancária. Conforme ilustrado na Figura 5 a seguir, essas funcionalidades representam ações comuns em sistemas de software e foram projetadas para testar de maneira prática a solução proposta. Os dados gerados são armazenados em um banco de dados PostgreSQL, configurado em um *container* Docker, refletindo uma arquitetura amplamente utilizada no desenvolvimento de sistemas modernos.

A criação de múltiplos registros em um curto intervalo de tempo pode gerar sobrecarga na memória do banco de dados, especialmente em situações onde grandes volumes de dados são gravados rapidamente. Além disso, operações de sumarização, que utilizam funções de agregação como agrupamento e soma, exigem alto poder de processamento, especialmente em bases de dados com grandes volumes de informações. As documentações de sistema de banco de dados geralmente abordam diversos mecanismos para evitar sobrecargas, como criação de indexes (índices), utilização de comandos explain (explicar) para analisar e melhorar desempenho de comandos SQL, entre outros. Isto destaca os desafios associados ao gerenciamento de memória e processamento em transações intensivas e em cargas massivas de dados. Esses tipos de operações podem aumentar significativamente os tempos de resposta e ocasionar erros, como timeouts, ou até mesmo falhas sistêmicas, dependendo dos recursos de infraestrutura disponíveis e da configuração aplicada.

Figura 5 – Diagrama de caso de uso.



Fonte: Do autor (2024).

O escopo desenvolvido não apenas facilita a criação de um ambiente controlado para os testes, mas também permite avaliar os impactos do *rate limit* em operações críticas, como as mencionadas acima. Embora tenha sido projetado para este estudo em específico, a solução proposta de *rate limit* possui flexibilidade e pode ser adaptada para outros contextos e escopos, atendendo às necessidades de sistemas com diferentes requisitos e níveis de complexidade.

4.3.2 Arquitetura

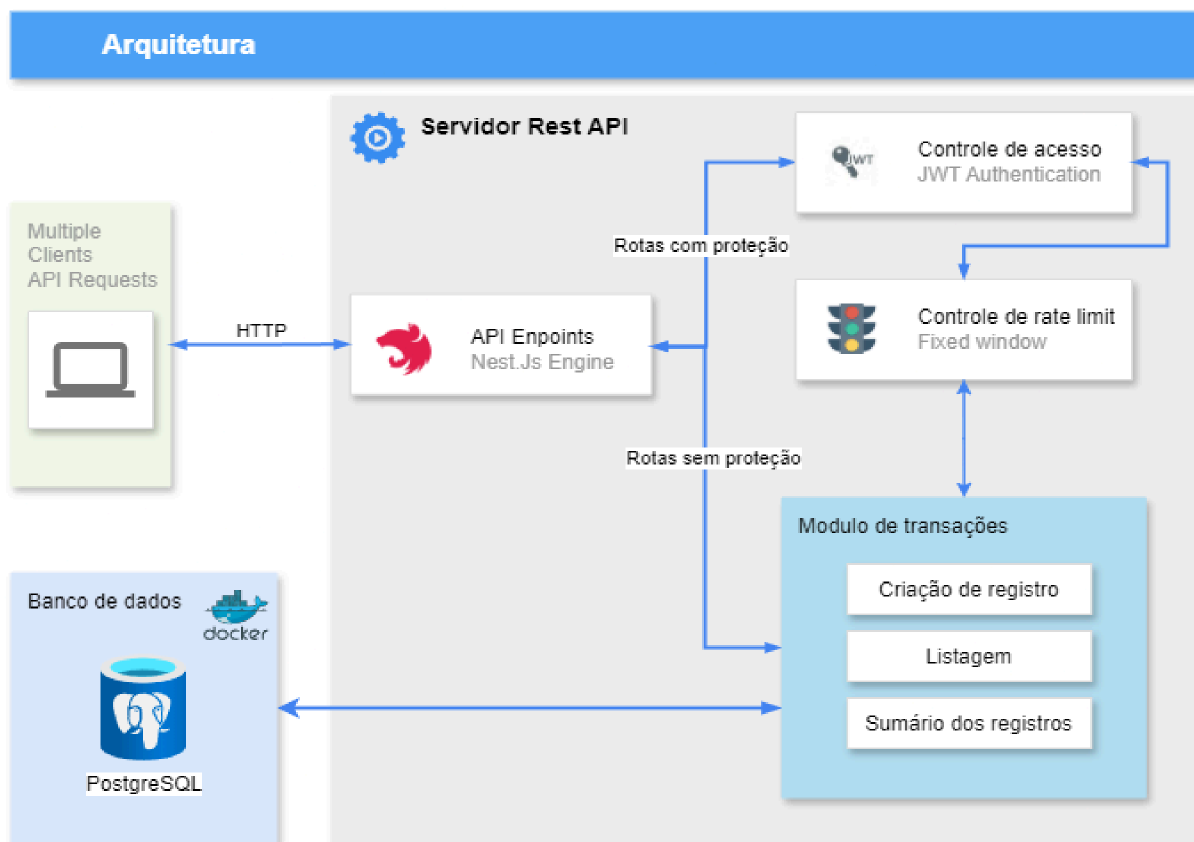
A arquitetura de software foi planejada para aproveitar o módulo de transações de maneira flexível, permitindo que ele seja utilizado com ou sem os mecanismos de autenticação e *rate limit*. Essa abordagem garante que as mesmas funcionalidades sejam testadas sem interferências no código, eliminando possíveis variações que poderiam comprometer a análise de desempenho e qualidade das rotas.

O primeiro passo no desenvolvimento foi a criação da API utilizando o *framework* NestJS, escolhido por sua facilidade de implementação e recursos como módulos, provedores e *guards*, que foram fundamentais para a construção da solução. A modularidade nativa do NestJS permitiu a separação clara entre os diferentes componentes, facilitando o gerenciamento e a evolução do sistema.

Em seguida, foi desenvolvido o módulo de autenticação, utilizando o sistema JWT (JSON Web Token) devido à sua simplicidade e robustez em termos de segurança. Cada usuário possui um nome de usuário e senha, que são utilizados na rota de login. Em caso de autenticação bem-sucedida, um token com validade de 5 minutos é gerado e pode ser utilizado para acessar as demais rotas protegidas até sua expiração.

O módulo de transações foi implementado utilizando a biblioteca Sequelize, um ORM (*Object-Relational Mapping*). ORMs são ferramentas que permitem mapear objetos da aplicação para tabelas do banco de dados, simplificando a manipulação de dados e abstraindo a complexidade das consultas SQL. Esse módulo foi projetado para realizar operações como criação de registros, consulta de registros individuais ou em lote e agrupamento de dados. Para cada uma dessas operações, foram disponibilizadas duas rotas: uma protegida por autenticação e limitação de taxa de requisições, e outra sem proteções, permitindo uma comparação direta nos testes. Por fim, foi criado o módulo de *rate limit*, cuja implementação será detalhada no próximo subcapítulo.

Figura 6 – Diagrama da arquitetura da aplicação proposta



Fonte: Do autor (2024).

A Figura 6, apresentada acima, ilustra a arquitetura descrita acima, destacando a integração entre os principais módulos, como autenticação, *rate limit* e transações, além da interação com o banco de dados PostgreSQL configurado em um *container* Docker. Essa estrutura modular e flexível reflete uma arquitetura amplamente utilizada em sistemas modernos, permitindo testes detalhados e comparativos de desempenho e segurança.

4.3.2 Rate Limit

A implementação do *rate limit* nesta aplicação segue o algoritmo de Janela Fixa (*Fixed Window*). A escolha do algoritmo de janela fixa deve-se à sua simplicidade de implementação, facilidade de interpretação dos resultados nos testes, eficiência no uso de memória e adequação ao objetivo de validar, de forma clara e objetiva, os impactos do *rate limit* na estabilidade e disponibilidade da API.

Figura 7 – Código fonte da configuração do *rate limit*

```
const userId = user.userId;
const route = request.route.path;
const rateLimiterKey = `rate-limiter-${userId}-${route}`;

let rateLimiterConfig = this.rateLimiters.get(rateLimiterKey);

if (!rateLimiterConfig) {
  const userRateLimit = this.cacheService.get<{ points: number; duration: number }>(`rate-limit-${userId}`);
  const routeRateLimit = this.cacheService.get<{ points: number; duration: number }>(`rate-limit-${route}`);

  const points = userRateLimit?.points || routeRateLimit?.points || DEFAULT_RATE_LIMIT_POINTS;
  const duration = userRateLimit?.duration || routeRateLimit?.duration || DEFAULT_RATE_LIMIT_DURATION;

  const rateLimiter = new RateLimiterMemory({
    points,
    duration,
  });

  rateLimiterConfig = { rateLimiter, points, duration };
  this.rateLimiters.set(rateLimiterKey, rateLimiterConfig);
}
```

Fonte: Do autor (2024).

A Figura 7 acima, demonstra a configuração inicial e lógica de funcionamento da configuração do RL. Um valor padrão de requisições permitidas é definido para todas as rotas e usuários, garantindo um controle básico de acessos. No entanto, essa configuração é flexível, permitindo ajustes específicos por usuário ou rota, quando necessário. Cada consumo de requisição (*points*) é registrado na memória com um tempo de expiração (*duration*) configurado. Quando o tempo da janela expira, o número de requisições realizadas é automaticamente resetado, garantindo que o controle seja atualizado de forma contínua.

Para cada nova requisição realizada em uma rota protegida pelo *rate limit*, é feita uma tentativa de consumir um "ponto" do total de requisições permitidas dentro da janela configurada. Se a requisição for aceita, a rota é executada normalmente. No entanto, caso o número de requisições exceda o limite disponível, a API retorna um status code 429 com a mensagem "*Too Many Requests*" (em português, "Excesso de Requisições"). Esse tratamento de erro foi projetado para comunicar claramente o bloqueio ao cliente, facilitando a identificação do motivo. A implementação apresentada na Figura 8 demonstra os blocos de código que validam e aplicam essas funcionalidades, garantindo a proteção contra abusos e a manutenção da estabilidade do sistema.

Figura 8 – Código fonte validação dos pontos de *rate limit*

```
try {
  // Tente consumir um ponto do rate limiter
  const rateLimiterRes = await rateLimiterConfig.rateLimiter.consume(userId);

  this.setRateLimitHeaders(response, rateLimiterConfig, rateLimiterRes);
  this.setRateLimitInfoInRequest(request, rateLimiterConfig, rateLimiterRes);

  return true;
} catch (rejRes) {
  // Caso não haja pontos suficientes para consumir, retorne erro 429
  this.setRateLimitHeadersWithRetry(response, rateLimiterConfig, rejRes);

  response.status(HttpStatus.TOO_MANY_REQUESTS).json({
    success: false,
    limit: rateLimiterConfig.points,
    remaining: 0,
    reset: new Date(Date.now() + rejRes.msBeforeNext).toISOString(),
  });

  throw new HttpException('Too many requests', HttpStatus.TOO_MANY_REQUESTS);
}
```

Fonte: Do autor (2024).

Além disso, para melhorar a experiência dos usuários e sistemas que consomem a API, foram incluídas informações detalhadas nos headers e no corpo das respostas. Esses dados incluem:

- Número total de requisições permitidas por janela;
- Número remanescente de requisições dentro da janela atual;
- Data e hora exata da próxima atualização da janela;
- Tempo até a renovação da janela em milissegundos.

Essa abordagem fornece uma comunicação clara e estruturada para os clientes da API, permitindo um controle mais eficaz e previsível dos recursos.

4.3.3 Banco de dados e geração dos dados

Na criação da instância do banco de dados de forma isolada e controlada, foi utilizada uma instância do PostgreSQL configurada em um *container* Docker. Essa instância possui 1 vCPU e 512 MB de memória, permitindo simular um ambiente de recursos limitados. Na figura 9 abaixo, é possível observar a configuração do *container* e o status de uso de CPU e memória durante a execução.

Figura 9 – Código fonte instrumentalização do *container* em Docker

```
services:
  postgres:
    image: postgres:13
    container_name: postgres
    environment:…
    ports:
      - "5432:5432"
    volumes:
      - postgres_data:/var/lib/postgresql/data
    deploy:
      resources:
        limits:
          memory: 512M
          cpus: '1'
        reservations:
          memory: 256M
```

```
PROBLEMAS   SAÍDA   CONSOLE DE DEPURAÇÃO   TERMINAL   PORTAS   GITLENS   COMENTÁRIOS

CONTAINER ID   NAME       CPU %   MEM USAGE / LIMIT    MEM %   NET I/O           BLOC
b0767cf94f13   postgres   0.03%   50.86MiB / 512MiB    9.93%   43.3kB / 123kB    37.2
```

Fonte: Do autor (2024).

Para a criação e gestão das tabelas, bem como para a geração dos dados simulados, utilizou-se o Sequelize, uma ferramenta ORM (*Object-Relational Mapping*) que facilita a manipulação do banco de dados através de scripts automatizados. Esse processo foi essencial para garantir a replicação fácil e consistente do ambiente em diferentes cenários de teste.

Os comandos utilizados:

- Criação das tabelas: npx sequelize-cli db:migrate --config sequelize-config.js;
- Geração dos dados: npx sequelize-cli db:seed:all --config sequelize-config.js.

A Figura 10 ilustra a estrutura das tabelas criadas e os scripts de inserção de dados. A tabela foi projetada com campos essenciais, como id, account, description, date e value, que refletem registros financeiros comuns, garantindo um cenário próximo ao real para execução dos testes.

Figura 10 – Código fonte *migrations* e *seeders* para geração da tabela e dados no banco de dados

```
up: async (queryInterface, Sequelize) => {
  await queryInterface.createTable('MOCK_DATA', {
    id: {
      type: Sequelize.INTEGER,
      allowNull: false,
      primaryKey: true,
      autoIncrement: true
    },
    account: {
      type: Sequelize.INTEGER,
      allowNull: false
    },
    description: {
      type: Sequelize.STRING,
      allowNull: false
    },
    date: {
      type: Sequelize.DATE,
      allowNull: false
    },
    ind_dc: {
      type: Sequelize.STRING(1),
      allowNull: false
    },
    value: {
      type: Sequelize.DECIMAL(6, 2),
      allowNull: false
    },
```

```
'use strict';

module.exports = {
  up: async (queryInterface, Sequelize) => {
    await queryInterface.bulkInsert('MOCK_DATA', [
  {
    "account": 678901,
    "description": "Transaction",
    "date": "2024-03-31T00:07:56.603Z",
    "ind_dc": "D",
    "value": 211.38,
    "createdAt": "2024-11-15T18:42:35.650Z",
    "updatedAt": "2024-11-15T18:42:35.650Z"
  },
  {
    "account": 345678,
    "description": "Transaction",
    "date": "2024-07-17T11:36:11.135Z",
    "ind_dc": "C",
    "value": 636.38,
    "createdAt": "2024-11-15T18:42:35.650Z",
    "updatedAt": "2024-11-15T18:42:35.650Z"
  },
  {
    "account": 789012,
    "description": "Transaction",
    "date": "2024-05-20T01:40:10.890Z",
```

Fonte: Do autor (2024).

Durante o processo de inserção de dados, foi identificado que, ao tentar inserir 200.000 registros ou mais de uma só vez, a instância com 512 MB de memória apresentava erros devido à limitação de recursos de memória. Esse comportamento evidencia a sobrecarga gerada por operações massivas em ambientes com restrição de memória, resultando em erros relacionados à falta de capacidade para processar a demanda.

A análise reforça a importância de considerar a configuração dos recursos alocados para o banco de dados em cenários de alta carga, além de destacar a necessidade de estratégias de controle, como divisão das inserções em lotes menores ou ajustes na infraestrutura, para garantir a estabilidade e o desempenho do sistema. Por isso, foram realizadas quatro rodadas de inserções de 150.000 registros cada, totalizando 600.000 registros, como forma de contornar o problema e assegurar o sucesso da criação de um banco de dados com milhares de registros para a simulação.

5 TESTES E ANÁLISE DOS RESULTADOS

Este capítulo apresenta os testes realizados e a análise dos resultados obtidos na aplicação da solução proposta no capítulo anterior. A etapa de testes, seguida pela análise dos resultados, desempenha um papel fundamental na validação da eficácia do sistema de *rate limit* implementado, contribuindo para garantir a estabilidade e a qualidade da API. Além disso, essa fase permite identificar possíveis aprimoramentos no modelo proposto, fornecendo percepções valiosas para a governança e proteção das APIs em cenários de alta demanda e uso abusivo.

5.1 Validação do Sistema de Rate Limit

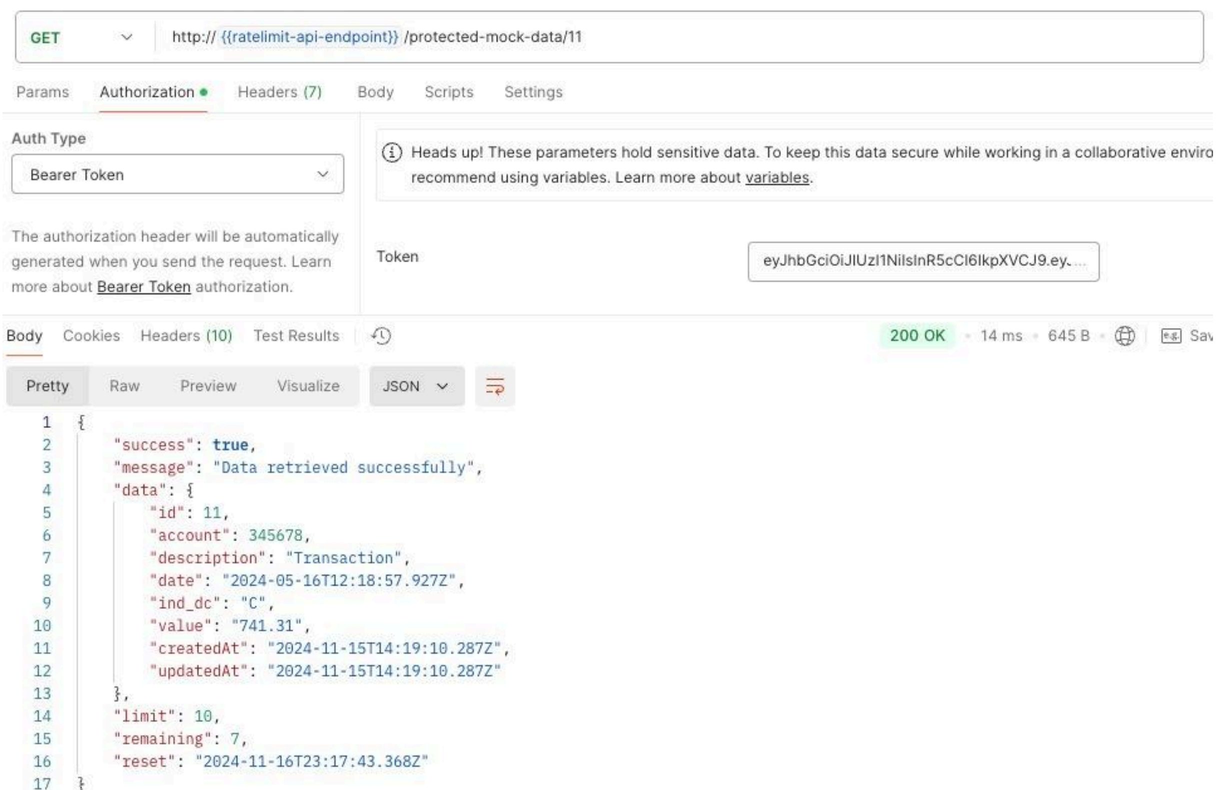
Neste subcapítulo, são apresentados os testes realizados para validar o funcionamento do sistema de *rate limit* nas rotas protegidas. O objetivo principal é demonstrar como o sistema limita o número de requisições por unidade de tempo, seguindo as políticas configuradas, e verificar o comportamento da API ao atingir os limites estabelecidos. Para isso, foram utilizados o Postman para realizar as requisições e a verificação dos resultados.

5.1.1 Limitação de requisições por unidade de tempo

Em cenários normais, o sistema permite até 10 requisições dentro de uma janela de tempo configurada. Conforme ilustrado na Figura 11 seguinte, requisições

válidas retornam uma resposta de sucesso com código de status 200 e incluem informações adicionais nos *headers* e no corpo da resposta.

Figura 11 – Requisição via Postman para rota com *rate limit* com sucesso

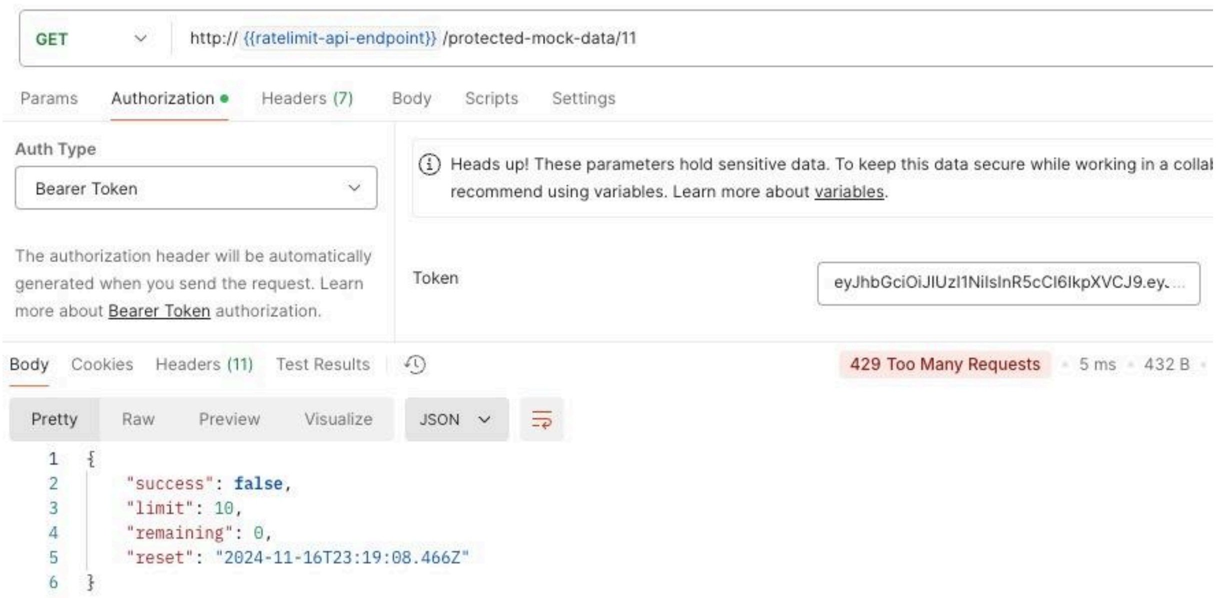


Fonte: Do autor (2024).

5.1.2 Resposta do sistema ao exceder o limite de taxa

Ao exceder o número de requisições permitidas, o sistema retorna código de status 429 (*Too Many Requests*), indicando que o limite foi atingido. Na Figura 12 abaixo, observa-se que o corpo da resposta informa o limite configurado (*limit*), o número de requisições restantes (*remaining* igual a 0) e o tempo exato para a próxima renovação da janela (*reset*).

Figura 12 – Requisição via Postman para rota com *rate limit* ao exceder o limite configurado



Fonte: Do autor (2024).

Os testes realizados validaram com sucesso o funcionamento do sistema de *rate limit*, demonstrando:

- Eficiência na limitação: O sistema bloqueia adequadamente as requisições que excedem o limite configurado, garantindo a estabilidade da API;
- Transparência nas respostas: Informações detalhadas são fornecidas nas respostas, facilitando o diagnóstico por parte dos usuários e integradores;
- Conformidade com políticas: O comportamento observado está em total conformidade com as políticas de *rate limit* definidas, reforçando a eficácia da solução.

5.2 Cenários de Testes de Carga

Nesta seção, são descritos os cenários de teste simulados para avaliar a performance da API em condições de alta demanda. O objetivo desses testes é analisar o comportamento da API em diferentes configurações, com e sem a aplicação de *rate limit*, utilizando ferramentas de teste de carga para gerar alto

throughput (volume) de requisições simultâneas. Além disso, os resultados dos dois cenários serão comparados com o intuito de:

- **Demonstrar a eficiência do *rate limit***: Evidenciar como o controle de requisições pode melhorar a estabilidade e a disponibilidade da API;
- **Identificar padrões de uso e sobrecarga**: Avaliar o comportamento da API em condições extremas e como o *rate limit* mitiga os riscos do sistema atingir limites críticos;
- **Estabelecer boas práticas**: Fornecer insights sobre a importância de aplicar *rate limit* em APIs para proteger recursos e melhorar a experiência do usuário.

Para ambos os cenários, foram realizadas simulações semelhantes, onde múltiplos usuários geraram um elevado número de requisições simultâneas, totalizando 50.000 ou mais requisições HTTP em um intervalo de aproximadamente 1 minuto. Todas as rotas do módulo de transações foram utilizadas nos testes, abrangendo operações de listagem de registros, criação de novos registros e sumarização de dados. As métricas analisadas incluíram taxa de sucesso/falha das requisições, latência ou tempo de resposta das requisições e uso de recursos como CPU e memória do banco de dados. A principal diferença entre os cenários foi a aplicação do *rate limit* no segundo cenário, com um total de cem usuários simultâneos realizando aproximadamente 500 requisições cada para totalizar aproximadamente as 50.000 requisições realizadas no cenário sem *rate limit*. Dessa maneira, permitindo uma comparação direta entre o comportamento da API sem restrições e com as limitações configuradas. Essa abordagem possibilitou avaliar o impacto do *rate limit* na estabilidade, eficiência e proteção do sistema contra sobrecargas.

5.3 Processo de Coleta dos Dados

Este subcapítulo detalha o processo técnico utilizado para a coleta e organização dos dados obtidos durante os testes de carga. As etapas e ferramentas empregadas foram cuidadosamente configuradas para garantir a precisão e a integridade dos dados, permitindo uma análise completa e confiável.

5.3.1 Utilização do k6

Os testes de carga foram realizados utilizando o k6, uma ferramenta de código aberto para simulação de requisições em larga escala. Foram criados scripts personalizados que geram operações em todas as rotas do módulo de transações, incluindo listagem, criação de registros e sumarização de dados. Esses scripts foram configurados para incluir valores randômicos em parâmetros como *offset* e *limit*, simulando cenários realistas de uso. O k6 foi responsável por coletar métricas relacionadas ao desempenho das requisições e ao comportamento da API sob alta demanda.

Figura 13 – Exemplo dos dados coletados pelo k6

```
execution: local
   script: load-test-mock-data.js
   output: -

scenarios: (100.00%) 1 scenario, 30 max VUs, 50s max duration (incl. graceful stop):
         * default: Up to 30 looping VUs for 20s over 1 stages (gracefulRampDown: 30s, gracefulStop: 30s)


data_received..................: 6.9 MB 329 kB/s
data_sent......................: 419 kB 20 kB/s
http_req_blocked...............: avg=8.69µs   min=0s       med=2µs      max=909µs    p(90)=7µs      p(95)=10µs
http_req_connecting............: avg=3.91µs   min=0s       med=0s       max=669µs    p(90)=0s       p(95)=0s
http_req_duration..............: avg=27.37ms  min=942µs    med=11.78ms  max=277.95ms p(90)=96.14ms  p(95)=107.93ms
  { expected_response:true }...: avg=27.37ms  min=942µs    med=11.78ms  max=277.95ms p(90)=96.14ms  p(95)=107.93ms
http_req_failed................: 0.00%  0 out of 2052
http_req_receiving.............: avg=38.1µs   min=7µs      med=25µs     max=799µs    p(90)=79µs     p(95)=104.44µs
http_req_sending...............: avg=12.37µs  min=2µs      med=9µs      max=210µs    p(90)=23µs     p(95)=33µs
http_req_tls_handshaking.......: avg=0s       min=0s       med=0s       max=0s       p(90)=0s       p(95)=0s
http_req_waiting...............: avg=27.32ms  min=925µs    med=11.72ms  max=277.89ms p(90)=96.1ms   p(95)=107.82ms
http_reqs......................: 2052   98.650123/s
iteration_duration.............: avg=610.42ms min=523.05ms med=543.04ms max=1.09s    p(90)=903.59ms p(95)=992.6ms
iterations.....................: 513    24.662531/s
success_requests...............: 2052   98.650123/s
vus............................: 29     min=2         max=29
vus_max........................: 30     min=30        max=30
```

Fonte: Do autor (2024).

A Figura 13 acima, um exemplo dos dados coletados pelo k6 ao término dos testes e como eles são apresentados. Entre os dados exibidos, destacam-se aqueles mais relevantes para este trabalho: a quantidade total de requisições HTTP realizadas (*http_reqs*), o número e percentual de requisições com falha (*http_req_failed*), o tempo de resposta das requisições (*http_req_duration*) e a linha que especifica as requisições bem-sucedidas (*expected_response:true*), permitindo diferenciar os tempos de resposta em casos de sucesso e falha. Além disso, é possível criar contadores personalizados para rastrear códigos de status específicos

nas respostas, o que foi utilizado para monitorar requisições bloqueadas pela limitação de taxa e identificar erros internos do servidor.

5.3.1 Ferramentas auxiliares

Além do k6, foram utilizados scripts em Bash para auxiliar no monitoramento e registro do consumo de recursos do banco de dados PostgreSQL instanciado em um *container* Docker. Esses scripts captam e registram, em tempo real, dados como, consumo de CPU e uso de memória. A coleta de dados foi automatizada para rodar simultaneamente aos testes do k6, garantindo que métricas detalhadas fossem registradas durante toda a execução dos testes (Figura 14). Os dados coletados foram armazenados em arquivos de log e posteriormente processados para gerar gráficos e análises estatísticas apresentadas nos próximos capítulos.

Figura 14 – Script utilizado para executar os testes de carga do k6 e coletar dados do banco de dados

```
# Função para coletar dados de CPU e memória do contêiner
collect_stats() {
  while true; do
    TIMESTAMP=$(date +%Y-%m-%dT%H:%M:%S)
    STATS=$(docker stats --no-stream --format "{{.CPUPerc}},{{.MemPerc}}" $CONTAINER_NAME)
    echo "$TIMESTAMP,$STATS" >> $OUTPUT_FILE
    sleep $INTERVAL
  done
}
# Iniciar a coleta de dados em segundo plano
collect_stats &
MONITOR_PID=$!
cd ../test # Navegar até o diretório onde o arquivo de teste do K6 está localizado
sleep 10 # Aguardando 10 segundos para o monitoramento coletar dados antes de iniciar os testes
# Executar os testes do K6
k6 run load-test-mock-data.js &
K6_PID=$!
wait $K6_PID # Esperar o K6 terminar
sleep 10 # Aguardar 10 segundos para o monitoramento coletar dados após o término dos testes
kill $MONITOR_PID # Finalizar a coleta de dados após o término dos testes do K6
```

Fonte: Do autor (2024).

5.4 Análise dos resultados obtidos

A análise a seguir apresenta os resultados obtidos durante os testes realizados, comparando o desempenho da API nos cenários com e sem sua aplicação. A análise inclui métricas coletadas, erros observados e o impacto do *rate limit* na estabilidade e eficiência do sistema.

5.4.1 Resultados sem rate limit

Os testes realizados sem a aplicação de *rate limit* revelam um alto índice de erros, incluindo *timeouts*, *internal server errors* e *connection reset by peer*, como demonstrado na Figura 15. O k6 registra cerca de 50.000 requisições, das quais 82,84% falha devido à sobrecarga do sistema. A latência média das requisições de sucesso é de 13 segundos, com poucas das primeiras requisições respondendo em milissegundos, mas o percentil 90 (p90) das requisições respondidas com sucesso alcança 26,76 segundos, evidenciando a instabilidade do sistema sob alta carga e a perda de performance.

Figura 15 – Resultados do k6 nos testes de carga sem *rate limit*

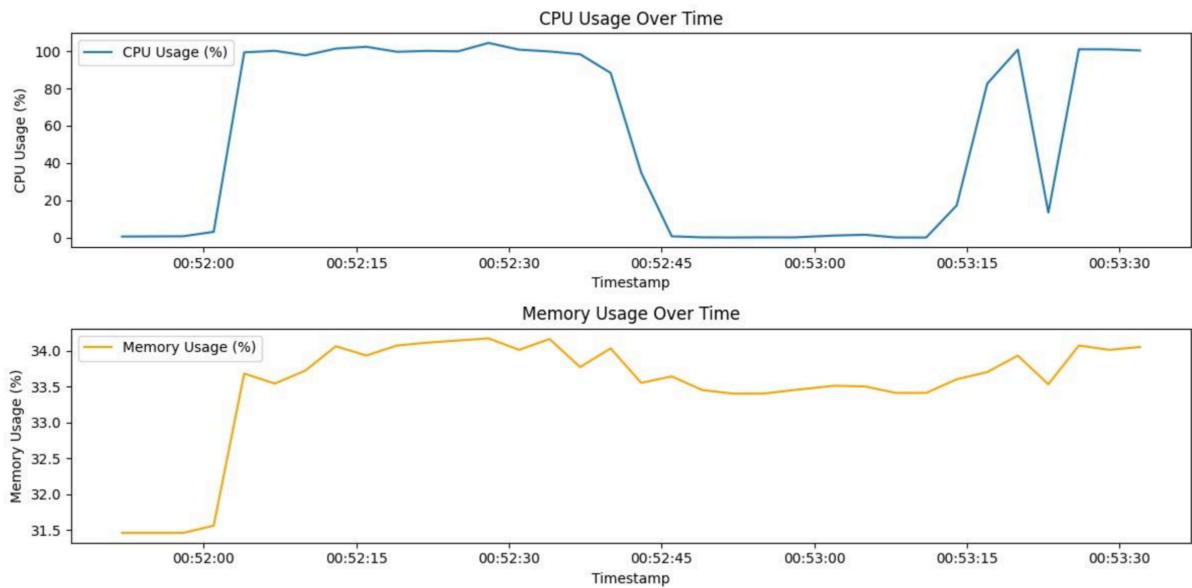
```
data_received..................: 74 MB  912 kB/s
data_sent......................: 9.6 MB 118 kB/s
faulty_requests................: 25918  319.120719/s
http_req_blocked...............: avg=99.05ms  min=0s       med=3µs    max=13.47s   p(90)=466.85ms p(95)=640.3ms
http_req_connecting............: avg=55.64ms  min=0s       med=0s     max=13.47s   p(90)=1.63ms   p(95)=524.39ms
http_req_duration..............: avg=13.96s   min=0s       med=5.11s  max=52.07s   p(90)=34.09s   p(95)=36.53s
  { expected_response:true }...: avg=13.12s   min=40.52ms  med=11.75s max=30.1s    p(90)=26.76s   p(95)=26.96s
http_req_failed................: 82.84% 42290 out of 51045
http_req_receiving.............: avg=356.07µs min=0s       med=0s     max=379.88ms p(90)=145µs    p(95)=335µs
http_req_sending...............: avg=227.72ms min=0s       med=11µs   max=17.1s    p(90)=129µs    p(95)=11.01ms
http_req_tls_handshaking.......: avg=0s       min=0s       med=0s     max=0s       p(90)=0s       p(95)=0s
http_req_waiting...............: avg=13.73s   min=0s       med=3.36s  max=52.07s   p(90)=34.09s   p(95)=36.53s
http_reqs......................: 51045  628.502088/s
iteration_duration.............: avg=29.93s   min=512.66ms med=19.52s max=1m12s    p(90)=1m3s     p(95)=1m6s
iterations.....................: 6554   80.697476/s
success_requests...............: 8755   107.797743/s
timed_out_requests.............: 16372  201.583626/s
vus............................: 4003   min=13             max=19980
vus_max........................: 20000  min=20000          max=20000
```

Fonte: Do autor (2024).

Durante o teste, a CPU do banco de dados opera consistentemente a 100% de uso, como ilustrado na Figura 16, mas há momentos de queda abrupta no consumo. Esses períodos coincidem com registros de erros de conexão no banco de

dados, indicando que a sobrecarga tornou o recurso indisponível por alguns segundos até sua normalização.

Figura 16 – Métricas de monitoramento do banco de dados nos testes de carga sem *rate limit*



Fonte: Do autor (2024).

Figura 17 – Logs de erro ao na API e ao conectar no banco de dados

```
PROBLEMAS   SAÍDA   CONSOLE DE DEPURAÇÃO   TERMINAL   PORTAS   GITLENS   COMENTÁRIOS
ERRO[0015] faulty_request read: connection reset by peer  source=console
ERRO[0015] faulty_request read: connection reset by peer  source=console
ERRO[0015] faulty_request read: connection reset by peer  source=console
```

```
rt:  ConnectionAcquireTimeoutError [SequelizeConnectionAcquireTimeoutError]: Operation timeout
ionManager.getConnection (/Users/rafael.siebeneichler/Desktop/GitHub/nest-api-ratelimit/ratelimi
ules/sequelize/src/dialects/abstract/connection-manager.js:294:48)
Ticks (node:internal/process/task_queues:60:5)
imeout (node:internal/timers:538:9)
.processTimers (node:internal/timers:512:7)
```

Fonte: Do autor (2024).

A Figura 17 acima apresenta exemplos desses erros, como *connection reset by peer* e *timeouts*, que ocorrem devido à incapacidade do banco em gerenciar o volume de conexões simultâneas durante os momentos críticos devido a sobrecarga comentada previamente.

5.4.2 Resultados com rate limit

Nos testes realizados com a aplicação de *rate limit*, a quantidade total de requisições geradas permanece similar ao cenário anterior, mas os resultados evidenciam uma melhoria significativa na estabilidade. Os erros observados são exclusivamente relacionados ao código de status 429 (*Too Many Requests*), conforme esperado pelas políticas de *rate limit* configuradas, ou seja, somente relacionados a usuários que estavam abusando do limite estabelecido. A latência das requisições de sucesso mantém-se estável em milissegundos, mesmo sob alta demanda, conforme ilustrado na Figura 18.

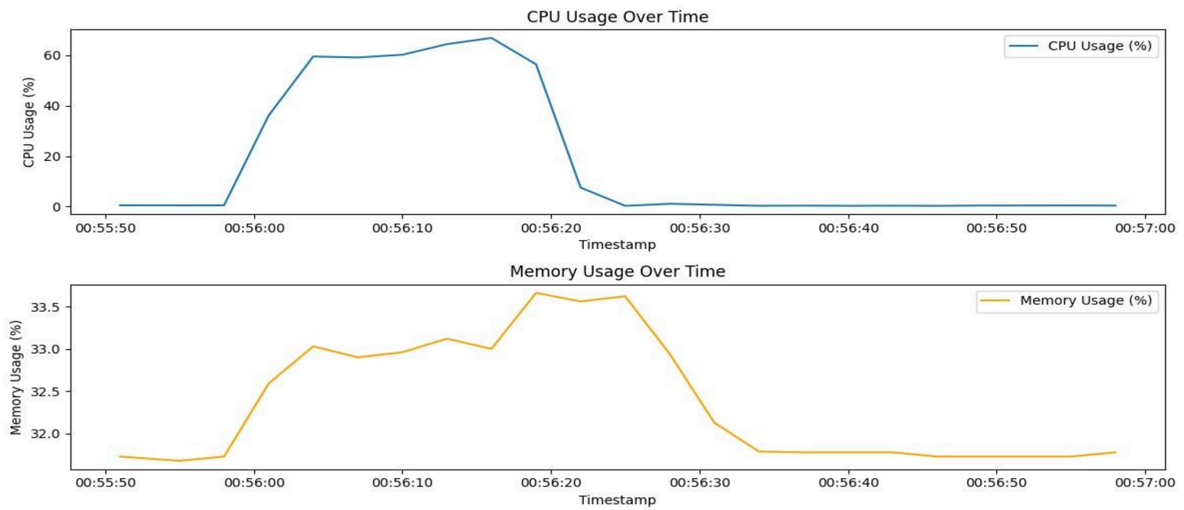
Figura 18 – Resultados do k6 nos testes de carga com *rate limit*

```
data_received..................: 34 MB  679 kB/s
data_sent......................: 19 MB  375 kB/s
http_req_blocked...............: avg=2.91µs   min=0s       med=1µs      max=2.39ms   p(90)=2µs      p(95)=3µs
http_req_connecting............: avg=1.16µs   min=0s       med=0s       max=2.29ms   p(90)=0s       p(95)=0s
http_req_duration..............: avg=5.78ms   min=386µs    med=2.21ms   max=96.23ms  p(90)=15.97ms  p(95)=20.27ms
  { expected_response:true }...: avg=7.84ms   min=1.14ms   med=3.55ms   max=96.23ms  p(90)=14.93ms  p(95)=19.1ms
http_req_failed................: 94.02% 47232 out of 50232
http_req_receiving.............: avg=15.07µs  min=5µs      med=11µs     max=3.55ms   p(90)=21µs     p(95)=30µs
http_req_sending...............: avg=5.67µs   min=1µs      med=4µs      max=2.78ms   p(90)=9µs      p(95)=12µs
http_req_tls_handshaking.......: avg=0s       min=0s       med=0s       max=0s       p(90)=0s       p(95)=0s
http_req_waiting...............: avg=5.76ms   min=375µs    med=2.19ms   max=96.16ms  p(90)=15.95ms  p(95)=20.21ms
http_reqs......................: 50232  994.531719/s
iteration_duration.............: avg=523.59ms min=502.01ms med=509.85ms max=685.73ms p(90)=566.45ms p(95)=581.67ms
iterations.....................: 12558  248.63293/s
rate_limited_requests..........: 47232  935.135415/s
success_requests...............: 3000   59.396304/s
vus............................: 259    min=6              max=259
vus_max........................: 260    min=260            max=260
```

Fonte: Do autor (2024).

No banco de dados, a CPU alcança picos de uso de até 60%, mas retorna rapidamente a níveis normais, como demonstrado na Figura 19. A ausência de sobrecarga evita erros inesperados e mantém o sistema em um estado operacional saudável. Não são observados *timeouts* ou erros relacionados à conexão com o banco de dados, reforçando a eficácia da limitação de taxas de requisição em proteger a API e demais recursos de abusos e sobrecargas.

Figura 19 – Métricas de monitoramento do banco de dados nos testes de carga com *rate limit*



Fonte: Do autor (2024).

5.4.3 Sumarização dos resultados

Os testes mostram que a aplicação de *rate limit* é essencial para manter a estabilidade e o desempenho da API em condições de alta demanda. O controle de requisições reduz significativamente o impacto da sobrecarga no banco de dados e nos tempos de resposta, evitando falhas e proporcionando uma experiência mais consistente para os usuários.

Quadro 2 – Comparativo dos resultados dos testes

Métrica	**Sem *rate limit***	**Com *rate limit***
Total de Requisições	~50.000	~50.000
Erros Observados	*Timeouts, Internal Server Error, Connection Reset by Peer*	Apenas Status 429 - *Too Many Requests*
Latência Média	13 segundos	Milissegundos
Latência (p90) em requisições com sucesso	26,76 segundos	< 500ms
Uso de CPU do banco de dados (Pico)	100%	60%

Erros no Banco de Dados	Sim	Não

Fonte: Do autor (2024).

O Quadro 2, apresentado acima, ilustra de forma clara as diferenças no uso da técnica de *RL*. Observa-se que, além de uniformizar os erros relacionados a abusos de usuários, a aplicação de limites de taxa garante que as requisições bem-sucedidas mantenham um alto padrão de tempo de resposta. Enquanto as requisições com sucesso sem a aplicação da técnica apresentaram latências médias de 13 segundos, as requisições com *rate limit* foram processadas em milissegundos, representando um desempenho dezenas de vezes mais eficiente. Esses resultados evidenciam a eficácia da limitação de taxas de requisição em promover estabilidade, reduzir a sobrecarga e melhorar a qualidade do serviço prestado.

6 CONSIDERAÇÕES FINAIS

O presente trabalho teve como principal objetivo explorar e validar a aplicação de *rate limit* como uma ferramenta eficaz para a governança de APIs, visando melhorar a qualidade e disponibilidade enquanto mitiga problemas como sobrecargas, indisponibilidade e instabilidade, além de promover uma utilização mais justa e controlada dos recursos. A partir dos resultados obtidos, entende-se que o *rate limit* é uma solução viável e eficaz para mitigar os desafios apresentados, ainda há espaço para melhorias e adaptações que podem ser exploradas em futuros trabalhos.

Os testes realizados demonstraram que a aplicação de *rate limit* reduz significativamente os impactos negativos de cenários de alta demanda. No ambiente configurado, observou-se que, sem a aplicação do *rate limit*, das aproximadamente 50.000 requisições, 82,84% falham devido à sobrecarga do sistema, com uma latência média de 13 segundos e um percentil 90 (p90) de 26,76 segundos. O banco de dados demonstra o uso de 100% da CPU disponibilizada, queda brusca na utilização do recurso e são registrados erros no console da API.

Por outro lado, com a aplicação do limite de taxa de requisições permitidas por usuário e por rota, os erros observados ficam limitados ao código de status 429 (*Too Many Requests*), indicando que as requisições abusivas são controladas conforme o esperado. Além disso, a latência das requisições bem-sucedidas se mantêm em milissegundos, destacando a estabilidade da API. Esse resultado garante a continuidade do serviço e evita sobrecargas no banco de dados, que anteriormente causavam perda de performance no tempo de resposta das requisições com sucesso e erros críticos, como *timeouts* e falhas de conexão.

Outro ponto positivo foi a praticidade da implementação técnica da solução devido ao uso do NestJS, com sua arquitetura modular e o suporte nativo a *guards*, permitiu uma integração eficiente do *rate limit* com outros módulos, como autenticação e transações. Por outro lado, identificou-se que uma das principais dificuldades está em determinar com precisão a quantidade real de requisições que uma API pode suportar com base nos recursos de CPU e memória disponíveis. Essa dificuldade decorre de vários fatores como medição da complexidade da arquitetura, recursos de infraestrutura, concorrência e da complexidade do código fonte, entre outros. Assim, é ressaltado a necessidade de estudos mais detalhados sobre dimensionamento e alocação de recursos.

Apesar dos resultados obtidos, equilibrar limites de taxa rígidos com uma experiência de usuário satisfatória continua sendo uma tarefa desafiadora. As configurações muito restritivas podem frustrar usuários legítimos, enquanto limites generosos podem expor a API a riscos de abusos e ataques. Este trabalho lança as bases para futuras investigações que abordem esses desafios de forma mais detalhada. Com base nisto e nos testes realizados e resultados analisados, propõem-se as seguintes sugestões de melhorias e linhas de pesquisa futuras.

A exploração de limites de taxa de requisição com aprendizagem de máquina**,** implementar técnicas de aprendizagem de máquina para desenvolver uma gestão de limites de requisição automática e dinâmica. Essa abordagem permitiria que os limites se adaptassem automaticamente ao número de usuários ativos e aos recursos internos disponíveis, como CPU e memória, promovendo uma alocação mais eficiente e inteligente.

Integração com sistemas de monitoramento, ao integrar a solução com ferramentas de monitoramento para acompanhar os SLOs visando garantir que os serviços estejam em norma com os SLAs definidos seria um bom complemento ao trabalho trazendo maior maturidade e robustez para a solução. A implementação de planos de resposta e geração de incidentes quando os SLOs estiverem fora do esperado melhora o tempo de detecção e recuperação dos serviços garantindo maior confiabilidade no serviço.

Um estudo sobre medição e cobrança de consumo de APIs, ao realizar estudos sobre como medir e monetizar o consumo de APIs, incluindo a criação de

modelos de cobrança baseados no uso, que podem ajustar automaticamente o preço quando ocorre algum abuso aos recursos pode beneficiar empresas que desejam oferecer serviços escaláveis e lucrativos.

Integração com sistemas de cache, ao explorar a integração com sistemas de cache para otimizar o desempenho das APIs, pode-se reduzir o número de consultas a banco de dados e melhorar os tempos de resposta, especialmente em operações de leitura e sumarização de dados realizadas frequentemente. Dessa forma, possibilitando um número maior de requisições na janela de tempo definida no *rate limit*.

Aprofundamento em testes de carga automatizados, expandir o escopo dos testes de carga para incluir cenários mais variados e ferramentas complementares que simulem condições ainda mais próximas de ambientes reais, avaliando o impacto do *rate limit* em diferentes tipos de arquiteturas, como por exemplo arquiteturas distribuídas, arquiteturas serverless, entre outras.

Por fim, este trabalho destaca a eficiência e relevância do *rate limit* como um componente fundamental na governança de APIs. Proporcionando mais estabilidade e proteção contra abusos ou ataques, além de contribuir para uma melhor alocação de recursos e uma experiência de usuário mais consistente. Contudo, o sucesso da sua aplicação depende de configurações criteriosas e de uma visão estratégica que considere tanto as necessidades técnicas quanto as expectativas dos usuários. O desenvolvimento de abordagens mais inteligentes e adaptáveis pode consolidar ainda mais o *rate limit* como uma solução indispensável para APIs em um cenário de demanda crescente e complexidade tecnológica.

REFERÊNCIAS

BERNERS-LEE, T.; NIELSEN, H.; FIELDING, R. T. **Hypertext Transfer Protocol – HTTP/1.0**. RFC Editor, 1996. RFC 1945. (Request for Comments, 1945). Disponível em: https://www.rfc-editor.org/rfc/rfc1945.txt. Acesso em: agosto de 2024

BEYER, Betsy; JONES, Chris; PETOFF, Jennifer; MURPHY, Niall Richard (orgs.). **Site Reliability Engineering**: How Google Runs Production Systems. 1. ed. Sebastopol: O'Reilly Media, 2016.

CHERNY, Boris. **Programming TypeScript: Making Your JavaScript Applications Scale**. 1. ed. Sebastopol: O'Reilly Media, 2019.

CLOUDFLARE. **Famous DDoS Attacks**. Disponível em: https://www.cloudflare.com/pt-br/learning/ddos/famous-ddos-attacks/. Acesso em: 17 ago. 2024.

CORREA, Daniel; LIM, Greg. **Practical Nest.js: Develop clean MVC web applications.** Publicação independente, 30 jan. 2022.

DANTAS, Daniel Chaves Toscanos. **SOAP (Protocolo de acesso a objetos Simples).** 2007. Disponível em: https://www.gta.ufrj.br/grad/07_2/daniel/index.html. Acesso em: 18 de ago. de 2024.

DOCKER, Inc. **What is Docker?**. Disponível em: https://docs.docker.com/get-started/docker-overview/. Acesso em: 16 out. 2024.

DURANTE, Daniel. **Supercharging Node.js Applications with Sequelize:** Create high-quality Node.js apps effortlessly while interacting with your SQL database. 1. ed. Birmingham: Packt Publishing, 28 out. 2022.

ELMASRI, Ramez; NAVATHE, Shamkant B. **Sistemas de banco de dados**. 7. ed. São Paulo: Pearson, 2018.

FIELDING, Roy Thomas: **Architectural styles and the design of network-based software architectures**, 2000. https://ics.uci.edu/~fielding/pubs/dissertation/top.htm. Acesso em: 19 de ago. de 2024.

FIELDING, Roy T.; NOTTINGHAM, Mark; RESCHKE, Julian. **HTTP Semantics**. Internet Engineering Task Force (IETF). Request for Comments: 9110. June 2022. Disponível em: https://httpwg.org/specs/rfc9110.html#overview.of.status.codes. Acesso em: 20 de ago. 2024.

GIL, Antonio Carlos. **Métodos e técnicas de pesquisa social**. 6 ed. São Paulo: Atlas, 2008.

GRAFANA LABS. **k6 Documentation**. Disponível em: https://grafana.com/docs/k6/latest/. Acesso em: 18 ago. de 2024.

HALILI, Festim; RAMADANI, Erenis. **Web Services: A Comparison of Soap and Rest Services. Modern Applied Science**, 2018. Disponível em: https://www.academia.edu/36092997/Web_Services_A_Comparison_of_Soap_and_Rest_Services. Acesso em: 18 out. 2024.

HAMILTON, Thomas. **Postman Tutorial – How to use for API Testing?**. Disponível em: https://www.guru99.com/postman-tutorial.html. Atualizado em: 22 nov. 2024. Acesso em: 23 nov. 2024.

JOHNSSON, Dan Bergh; DEOGUN, Daniel; SAWANO, Daniel. **Secure by Design**. Shelter Island, NY: Manning Publications, 2019.

MADDEN, Neil. **API Security in Action**. Shelter Island, NY: Manning Publications, 2020.

MAJOLO, Jackson Diesel. **DataCache: Gestão e padronização na comunicação entre aplicações**. 2019. Trabalho de Conclusão de Curso (Bacharelado em Engenharia de Software) – Universidade do Vale do Taquari, Lajeado, 2019.

MALKI, Amine El; ZDUN, Uwe; PAUTASSO, Cesare. **Impact of API Rate Limit on Reliability of Microservices-Based Architectures**. 2022. Disponível em: https://ieeexplore.ieee.org/abstract/document/9912639. Acesso em: 18 ago. 2024.

MASSÉ, Mark: **REST API Design Rulebook**, volume 2012. O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol.

MILETO, Andrei Cioqueta. **Disponibilização de Dados de Fundos de Investimento através de uma API**. 2021. Trabalho de Conclusão de Curso (Bacharelado em Engenharia de Software) – Universidade do Vale do Taquari, Lajeado, 2021.

MYERS, Glenford J.; BADGETT, Tom; SANDLER, Corey. **The Art of Software Testing**. 3. ed. Hoboken, NJ: John Wiley & Sons, Inc., 2012.

NESTJS. **Guards**. Disponível em: https://docs.nestjs.com/guards. Acesso em: 24 out. 2024.

NPM, Inc. **About npm**. Disponível em: https://docs.npmjs.com/about-npm. Acesso em: 18 jun. 2024.

PANASIEWICZ, Roberlei; BAPTISTA, Paulo Agostinho N. **Metodologia da Pesquisa:** A ciência e seus métodos. Belo Horizonte. 2013. Universidade FUMEC. Disponível em: <http://ppg.fumec.br/ecc/wp-content/uploads/2016/12/MetodCientifica_02.pdf>. Acesso em: abril de 2024.

PATIDAR, Priya. **Introduction to API Rate Limiting:** Understanding the Basics and Its Importance. Medium, 28 jan. 2024. Disponível em: https://medium.com/the-developers-diary/introduction-to-api-rate-limiting-understanding-the-basics-and-its-importance-fde0b5af995b. Acesso em: 24 out. 2024.

POSTGRESQL. **What Is PostgreSQL?**. PostgreSQL Global Development Group. Disponível em: https://www.postgresql.org/docs/current/intro-whatis.html. Acesso em: 24 out. 2024.

PRESSMAN, Roger S. **Engenharia de Software:** Uma abordagem profissional. 7. ed. Porto Alegre: AMGH Editora Ltda., 2011.

SERBOUT, Souhaila; MALKI, Amine El; PAUTASSO, Cesare; ZDUN, Uwe. **API Rate Limit Adoption - A Pattern Collection**. 2023. Disponível em: https://dl.acm.org/doi/abs/10.1145/3628034.3628039. Acesso em: 18 ago. 2024.

SOMMERVILLE, Ian. **Engenharia de software**. 10. ed. São Paulo: Pearson, 2018.

STRÖHER, João Miguel. **Framework para Desenvolvimento e Implantação de Aplicações HTTP Utilizando Serverless Computing**. 2023. Trabalho de Conclusão de Curso (Bacharelado em Engenharia de Software) – Universidade do Vale do Taquari, 2023,

TOULAS, Bill. **Attacks abusing programming APIs grew over 600 percent in 2021**. Bleeping Computer, 2022. Disponível em: https://www.bleepingcomputer.com/news/security/attacks-abusing-programming-apis-grew-over-600-percent-in-2021/. Acesso em: 16 jun. 2024.

WHITEHEAD, Jim. **Oralidade e hipertexto: uma entrevista com Ted Nelson**. In. O hipertexto em tradução. RIBEIRO, Ana Elisa; COSCARELLI, Carla Viana (Org.). Belo Horizonte: FALE/UFMG, 2007.

WILKES, Maurice V.; WHEELER, David J.; GILL, Stanley. **The Preparation of Programs for an Electronic Digital Computer**. Cambridge: Addison-Wesley Press, 1951.

ZIMMERMANN, Olaf; STOCKER, Mirko; LÜBKE, Daniel; ZDUN, Uwe; PAUTASSO, Cesare. **Patterns for API Design:** Simplifying Integration with Loosely Coupled Message Exchanges. 1. ed. Boston: Pearson Education, 2023