UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA
CURSO DE SISTEMAS DE INFORMAÇÃO

Nicolas Nascimento Maso

Comparativo entre arquiteturas de APIs - REST, GraphQL e gRPC

Florianópolis - SC

2024

Nicolas Nascimento Maso

Comparativo entre arquiteturas de APIs - REST, GraphQL e gRPC

Trabalho de Conclusão do Curso de Graduação em Sistemas de Informação do Centro Tecnológico da Universidade Federal de Santa Catarina como requisito para a obtenção do título de Bacharel em Sistemas de Informação.

Orientador: Prof. José Eduardo de Lucca

Florianópolis - SC

2024

Ficha catalográfica gerada por meio de sistema automatizado gerenciado pela BU/UFSC.

Dados inseridos pelo próprio autor.

Maso, Nicolas Nascimento
Comparativo entre arquiteturas de APIs - REST, GraphQL e gRPC / Nicolas Nascimento Maso ; orientador, José Eduardo de Lucca, 2024.
80 p.

Trabalho de Conclusão de Curso (graduação) - Universidade Federal de Santa Catarina, Centro Tecnológico, Graduação em Sistemas de Informação, Florianópolis, 2024.

Inclui referências.

1. Sistemas de Informação. 2. API. 3. REST. 4. GraphQL. 5. gRPC. I. de Lucca, José Eduardo. II. Universidade Federal de Santa Catarina. Graduação em Sistemas de Informação. III. Título.

Nicolas Nascimento Maso

Comparativo entre arquiteturas de APIs - REST, GraphQL e gRPC

Este Trabalho de Conclusão de Curso foi julgado adequado para obtenção do título de Bacharel e aprovado em sua forma final pelo Curso Sistemas de Informação.

Local, dia de mês de ano.

_____

Coordenação do Curso

Banca examinadora

_____

Prof. José Eduardo de Lucca.
Orientador

_____

Prof. Elder Rizzon Santos
Membro da Banca Examinadora

_____

Prof. Cristian Koliver
Membro da Banca Examinadora

Florianópolis, 2024.

Dedico este trabalho a minha família que me deu todo o suporte necessário para que eu conseguisse chegar aonde estou hoje.

“O ontem é história, o amanhã é um mistério, mas o hoje é uma dádiva. É por isso que se chama presente.”

Mestre Oogway

RESUMO

Com o crescimento do mercado de tecnologia, a cada dia novas ideias surgem e novos sistemas e soluções são criadas para resolver um determinado problema. Mesmo com toda a informação e com as tecnologias disponíveis atualmente, muitos projetos são encerrados antes do prazo ou tem diversos problemas ao longo do seu desenvolvimento, devido a decisões que foram tomadas e que possivelmente não eram a melhor a ser feita naquele momento. Consequentemente, novos profissionais qualificados são requisitados para criar e dar manutenção nestas soluções. Além disso, diariamente novas tecnologias para desenvolvimento são criadas e também atualizadas, recebendo novas bibliotecas e frameworks, enquanto outras tecnologias que já dominaram o mercado vão sendo ultrapassadas. Nesse sentido, com pesquisas e o desenvolvimento de uma API, esse trabalho busca realizar uma análise comparativa entre três possíveis arquiteturas de APIs, sendo elas REST, GraphQL e gRPC, trazendo uma comparação entre as características, documentação, utilização, desempenho e outros tópicos, a fim de detalhar e destacar as diferenças de cada uma para que desenvolvedores e/ou arquitetos de softwares possam ter mais opções e então tomarem melhores decisões em seus projetos.

Palavras-chave: API; REST; GraphQL; gRPC; Back-end.

ABSTRACT

With the growth of the technology market, new ideas emerge every day and new systems and solutions are created to solve a particular problem. Even with all the information and technologies available today, many projects are terminated before the deadline or have various problems throughout their development, due to decisions that were made that were possibly not the best thing to do at the time. Consequently, new qualified professionals are required to create and maintain these solutions. In addition, new development technologies are created and updated on a daily basis, receiving new libraries and frameworks, while other technologies that have already dominated the market are being overtaken. In this sense, with research and the development of an API, this work seeks to carry out a comparative analysis between three API architectures, namely REST, GraphQL and gRPC, comparing their characteristics, documentation, use, performance and other topics, in order to detail and highlight the differences between each one so that software developers and/or architects can have more options and then make better decisions in their projects.

Keywords: API; REST; GraphQL; gRPC; Back-end.

LISTA DE SIGLAS E ABREVIATURAS

API	Application Programming Interface
REST	Representational State Transfer
GraphQL	Graph Query Language
gRPC	gRPC Remote Procedure Calls
HTTP	Hypertext Transfer Protocol
JSON	JavaScript Object Notation
URL	Uniform Resource Locator
SOAP	Simple Object Access Protocol
XML	Extensible Markup Language

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1. INTRODUÇÃO

Nos últimos anos o setor de TIC (Tecnologia da informação e comunicação) gerou milhares de empregos tanto a nível nacional como a nível mundial, e além disso, o setor possui previsão de muita demanda para os próximos anos e possui uma remuneração superior em comparação com outros setores. Segundo os últimos estudos realizados pela Brasscom (2021), que é a Associação das Empresas de Tecnologia da Informação e Comunicação e de Tecnologias Digitais, apenas no Brasil no ano de 2021, o macrossetor gerou mais de 198 mil empregos e em 2022 mais de 117 mil empregos. O mesmo estudo também aponta que em 2022, as ocupações relacionadas a desenvolvimento de sistemas representaram aproximadamente 70,7% de todas as ocupações do setor de TIC, e também que a remuneração média do profissional do macrossetor de TIC é 2,2 vezes superior ao salário médio nacional e até 2,8 vezes superior em relação ao subsetor de serviços de alto valor e software. No mesmo contexto, o estudo do ano de 2021 indica que haverá uma demanda de 797 mil profissionais até o ano de 2025 apenas no Brasil, dessa forma, a profissão de desenvolvedor de sistemas e aplicativos está cada vez mais em alta devido à grande demanda desse tipo de profissional para as empresas que necessitam dos mesmos para desenvolver e manter seus produtos e serviços no mundo cada vez mais digital.

1.1. JUSTIFICATIVAS

Conforme dados apresentados anteriormente, a área de desenvolvimento de software está com uma alta demanda de profissionais de tecnologia qualificados como arquitetos e desenvolvedores de software no mercado mundial, nesse sentido para se tornar um profissional qualificado é necessário aprender e saber usufruir das excelentes tecnologias disponíveis que temos atualmente.

O conceito de Application Programming Interface (API) será detalhado na Seção 2.1 do presente estudo, porém de forma sucinta, sua sigla em

português significa Interface de Programação de Aplicação, e possui um conjunto de rotinas e padrões que permitem a comunicação entre dois ou mais softwares, ou seja, fornece uma maneira padronizada para que sistemas acessem dados e funcionalidades de outros sistemas. As APIs desempenham um papel fundamental na integração de sistemas e no desenvolvimento de novas aplicações, e a escolha de uma arquitetura pode ter um impacto significativo na eficiência e no desempenho das soluções tecnológicas, podendo inclusive ter impacto na economia de recursos.

As arquiteturas de APIs Representational State Transfer (REST), Graph Query Language (GraphQL) e gRPC Remote Procedure Calls (gRPC) foram selecionadas para o desenvolvimento do presente estudo devido a sua popularidade na comunidade de desenvolvimento de software, que se deram principalmente devido a popularização da arquitetura de microsserviços, a qual abriu caminho para novas opções de comunicação entre aplicações.

Sendo assim, as tomadas de decisão realizadas ao se iniciar o desenvolvimento de uma API são importantes e de alta relevância para o mercado de tecnologia. Cada escolha pode gerar um impacto diferente, seja na eficiência, flexibilidade, economia e demais fatores de um projeto de desenvolvimento. Além disso, conteúdos similares ao do presente estudo são escassos na comunidade de desenvolvimento de software, e os que existem são, na grande maioria, superficiais, ou seja, um estudo mais detalhado seria de grande utilidade para auxiliar a comunidade de desenvolvimento de software em relação a desenvolvimento de APIs.

1.2. OBJETIVOS GERAIS

O presente trabalho tem como objetivo geral realizar uma análise e comparação entre três das principais arquiteturas de desenvolvimento de APIs utilizadas no mercado atualmente, sendo elas as arquiteturas REST, GraphQL e gRPC, em relação aos aspectos de facilidade de implementação, qualidade da

documentação, relevância na comunidade de software e o desempenho com as seguintes métricas:

- REST, GraphQL e gRPC
  - Quantidade de dados recebidos
  - Quantidade de dados enviados
  - Tempo para concluir uma iteração completa
  - Total de iterações

- Específico para REST e GraphQL
  - Tempo total de uma requisição
  - Tempo gasto para estabelecer a conexão TCP com o servidor
  - Tempo gasto no envio de dados para o servidor
  - Tempo gasto aguardando a resposta do servidor
  - Tempo gasto para receber dados de resposta do servidor
  - Total de solicitações de Hypertext Transfer Protocol (HTTP)

- Específico para gRPC
  - Tempo para receber resposta do servidor

Todos os aspectos que não estiverem relacionados a desempenho e não estiverem referenciados tecnicamente estarão sob o ponto de vista de um profissional de desenvolvimento de software e da comunidade de desenvolvimento.

1.3. OBJETIVOS ESPECÍFICOS

Os objetivos específicos deste trabalho são:

- Realizar um estudo teórico a respeito das arquiteturas REST, GraphQL e gRPC para desenvolvimento de APIs.

- Demonstrar as principais diferenças entre as arquiteturas REST, GraphQL e gRPC, apresentando suas características, configurações e limitações.
- Desenvolver uma API que contemple respectivamente as arquiteturas REST, GraphQL e gRPC e que possua funcionalidades com operações de escrita e leitura de dados.
- Realizar uma análise comparativa de configuração e desempenho com múltiplas quantidades de dados entre as arquiteturas REST, GraphQL e gRPC presentes na API desenvolvida no trabalho.

1.4. METODOLOGIA

Para o desenvolvimento deste trabalho foi adotada uma metodologia de pesquisa e desenvolvimento que foi dividida nas seguintes etapas:

1.4.1. Etapa 1 - Pesquisa dos conceitos fundamentais para o trabalho

Nesta etapa do projeto será realizado um estudo teórico na literatura com base em artigos, teses, vídeos e publicações a respeito dos principais conceitos e temáticas relacionadas que envolvem o respectivo trabalho, sendo os seguintes conceitos:

- API;
- Desenvolvimento front-end e back-end;
- REST;
- GraphQL;
- gRPC;
- Framework;

1.4.2. Etapa 2 - Estudo das arquiteturas

Além da pesquisa inicial com os conceitos básicos, para cada uma das arquiteturas principais que foram utilizadas como base para o estudo (REST, gRPC e GraphQL) será realizado um estudo mais aprofundado com leitura das

documentações oficiais, vídeos, artigos, notícias e estudos que detalham cada uma delas.

1.4.3. Etapa 3 - Planejamento da parte prática

Será realizado um planejamento definindo como o desenvolvimento da parte prática será realizado. Inicialmente está definido que será desenvolvido uma única API que possua os três estilos arquiteturais do estudo, sendo REST, gRPC e GraphQL, além também da definição dos casos de uso, requisitos da aplicação e das ferramentas e tecnologias que serão selecionadas para o desenvolvimento da API.

1.4.4. Etapa 4 - Desenvolvimento da API

Essa etapa será a responsável pelo início e finalização do desenvolvimento da API do estudo em questão. Será realizado a implementação da API com a configuração das três arquiteturas do estudo. A API será implementada com todos os casos de uso e com os requisitos do planejamento previsto na etapa 3.

1.4.5. Etapa 5 - Testes e análise com a API

Após a conclusão do desenvolvimento da API, serão definidos os testes por qual a API será submetida e que os resultados serão utilizados para a conclusão do estudo. Os testes serão realizados com base nos casos de uso da aplicação e com diferentes quantidades de dados, sendo que a API consumirá uma base de dados com milhares de dados pré cadastrados.

1.4.6. Etapa 6 - Conclusão

Com a finalização dos testes será desenvolvido a conclusão do estudo do trabalho com base nos resultados gerados pelos mesmos, a fim de finalizar a análise e comparação das três arquiteturas de APIs estudados no presente trabalho. O trabalho completo será disponibilizado, incluindo a dissertação e o código da parte prática.

2. CONCEITOS BÁSICOS

2.1. DESENVOLVIMENTO FRONT-END E BACK-END

No âmbito de desenvolvimento de sistemas, existem algumas nomenclaturas que são utilizadas para diferenciar e classificar a especialização de um desenvolvedor e que fazem referência a qual parte da estrutura do código é desenvolvida pelo mesmo. O termo *front-end* remete a parte de uma aplicação que os usuários veem e inclui elementos visuais como botões, caixas de seleção, gráficos e mensagens de texto, ou seja, é a parte que permite que os usuários interajam com a aplicação, enquanto que o termo *back-end* consiste na parte dos dados e na infraestrutura que fazem uma aplicação funcionar, de modo geral, realiza o armazenamento e o processamento dos dados da aplicação para os usuários (AWS, 2023).

2.2. FRAMEWORK

No cenário de desenvolvimento de software, um *framework* é uma estrutura que é utilizada para criar um software, essa estrutura funciona como uma base para que o desenvolvedor não precise lidar com a criação de lógica extra desnecessária desde o início. Essa estrutura é semelhante a um *template*, pois o desenvolvedor pode modificá-la e adicionar determinados recursos e funcionalidades superiores para criar um projeto complexo e amplo que pode ser usado por muitas pessoas (OLAWANLE, 2022). Os *frameworks* são geralmente

associados a linguagens de programação específicas e também são apropriados para várias tarefas, dependendo da situação. Existem frameworks de *back-end* para tarefas de *back-end*, *frameworks* de *front-end* para tarefas de *front-end*, *frameworks* de dispositivos móveis para tarefas de dispositivos móveis e muitos outros.

2.3. API

O conceito de API não possui uma única definição no cenário de desenvolvimento de software. Segundo a plataforma AWS (2023b), uma das maiores plataformas de serviços de computação em nuvem, e seguindo o contexto apresentado anteriormente, uma interface serve como um contrato de serviço entre duas ou mais aplicações e uma aplicação refere-se a qualquer software que possua alguma função distinta, ou seja, uma API é um contrato que irá definir como duas ou mais aplicações irão se comunicar utilizando solicitações e respostas. Já para a empresa Red Hat (2023), uma das empresas líder mundial em soluções empresariais *open source* (código aberto) como o Linux, as APIs são conjuntos de ferramentas, definições e protocolos para a criação de aplicações de software.

Uma analogia interessante e muito utilizada para o conceito de API é o do funcionamento de um pedido em um restaurante. O menu lista os pratos disponíveis e as opções de personalização, como adicionar ingredientes extras ou solicitar que um prato seja preparado de uma determinada maneira. O cliente faz o pedido ao garçom, que o leva para a cozinha. A cozinha, por sua vez, segue as instruções do menu para preparar o prato solicitado e devolve ao garçom, que entrega ao cliente. Da mesma forma, uma API fornece uma lista de operações e recursos disponíveis (como obter informações de um servidor web) e permite que um aplicativo faça solicitações usando essas operações e recursos. A API então lida com a solicitação e retorna os resultados ao aplicativo que fez a solicitação (HOSTGATOR, 2020).

Além disso, no âmbito de desenvolvimento para a Web, a maioria da comunicação é realizada pelo protocolo HTTP, que significa em português

Protocolo de Transferência de Hipertexto, esse protocolo é a base da internet e é usado para carregar as páginas web usando links de hipertexto.

Nesse contexto, há diversas tecnologias e possíveis técnicas que podem ser utilizadas para se desenvolver uma API. Para muitos desenvolvedores de software, como Dellano (2023) e Al-Khamisi (2023), não existe a melhor e nem a mais correta arquitetura de API, a escolha por parte do arquiteto ou do desenvolvedor da aplicação irá depender do escopo e requisitos do projeto, linguagem de programação que será utilizada, escalabilidade, documentação, desempenho, custo e demais fatores que podem impactar no desenvolvimento de uma solução.

3. ESTUDOS REALIZADOS

Para o embasamento teórico e prático do presente trabalho, foram realizadas pesquisas nas plataformas Google Acadêmico e IEEE (Institute of Electrical and Electronics Engineers) Xplore, utilizando as palavras-chave “API”, “REST”, “RESTful” “GraphQL”, “Google RPC” e “gRPC” e também realizando as combinações “REST API”, “GraphQL API”, “gRPC API”, “REST vs GraphQL”, “REST vs gRPC”, “GraphQL vs gRPC” e “REST GraphQL gRPC”. A Tabela 1 e Tabela 2 mostram a quantidade de resultados encontrados para cada plataforma. Os critérios definidos para seleção dos artigos foram:

- Estudos teóricos ou práticos que apresentem os conceitos de REST ou GraphQL ou gRPC e demais conceitos relacionados.

- Estudos teóricos ou práticos que realizaram testes e comparações de REST ou GraphQL ou gRPC, com alguma outra arquitetura de API.

Tabela 1 - Palavras-chave e resultados obtidos na plataforma Google Acadêmico

Palavra-chave	Total
“API”	6.350.000
“REST”	6.310.000
“RESTful”	1.040.000
“GraphQL”	6.760
“gRPC”	9.490
“Google RPC”	68.700
“REST API”	757.000
“GraphQL API”	5.330
“gRPC API”	5.070
“REST vs GraphQL”	5.160
“REST vs gRPC”	5.150
“GraphQL vs gRPC”	552
“REST GraphQL gRPC”	799

Tabela 2 - Palavras-chave e resultados obtidos na plataforma IEEE Xplore

Palavra-chave	Total
“API”	11.841
“REST”	21.231
“RESTful”	1.597
“GraphQL”	72
“gRPC”	78
“Google RPC”	16
“REST API”	621
“GraphQL API”	41
“gRPC API”	13
“REST vs GraphQL”	2
“REST vs gRPC”	0
“GraphQL vs gRPC”	0
“REST GraphQL gRPC”	1

Com base nos resultados das buscas, mesmo utilizando apenas duas plataformas é notável que estudos relacionados a API REST são encontrados com mais facilidade em relação a estudos com GraphQL e também gRPC, isso ocorre pois a arquitetura REST ainda domina o cenário de desenvolvimento de APIs, segundo pesquisa realizada por Postman (2023), cerca de 86% dos desenvolvedores utilizam a arquitetura REST para desenvolver suas APIs, sendo que 29% também utilizam GraphQL e aproximadamente 11% também utilizam gRPC.

Além da busca nas plataformas mencionadas, por meio da ferramenta Google Trends da Google, que é um dos maiores mecanismos de pesquisa na internet com 90.91% de participação no mercado mundial de mecanismos de busca segundo pesquisa realizada por Statcounter (2024), foi realizada uma comparação entre os termos de pesquisa ‘REST’, ‘gRPC’ e ‘GraphQL’ na categoria ciência da computação entre o período de 2004 até 2024 a nível mundial. O gráfico que foi gerado (Fig. 1) indicou o nível de interesse ao longo do tempo, quanto mais próximo de 100, maior o interesse. Foi possível notar o domínio do termo ‘REST’ até o presente momento, tendo o pico de interesse (100) em fevereiro de 2017 e um valor de 67 em abril de 2024. Já as arquiteturas GraphQL e gRPC tiveram respectivamente como valor máximo 32 (abril de 2022) e 18 (abril de 2022) e atualmente em abril de 2024 se encontram com 26 e 18 respectivamente.

Figura 1 - Gráfico Google Trends comparando os termos de pesquisa 'REST' (azul), 'GraphQL' (vermelho) e 'gRPC' (amarelo).

Após leitura de diversos estudos relacionados a APIs REST, GraphQL e gRPC, é notável que cada arquitetura para desenvolvimento de uma API possui suas características específicas, o que faz com que cada uma seja mais indicada ou que teve sua origem para casos de uso específicos.

Para que uma API possa ser considerada na arquitetura REST ela precisa ser desenvolvida seguindo os princípios REST, os quais serão detalhados na Seção 4.1 do presente estudo. Como REST é uma arquitetura e não está acoplada a nenhuma linguagem ou ferramenta, facilmente podemos encontrar diversas APIs REST públicas que possuem diversas diferenças. Segundo estudo realizado por Murphy et al. (2017), existem diversas APIs REST que não aplicam diretrizes de estilo (concebidas por grandes empresas de tecnologia, livros

conhecidos de design de API e fontes online) importantes no que diz respeito ao design da API. No estudo mencionado foram analisados 32 conjuntos de diretrizes publicamente disponíveis para APIs REST que foram comparadas entre diversas empresas e organizações, com o intuito de identificar as semelhanças e diferenças entre elas. Durante o estudo, os autores destacaram a importância da consistência e simplicidade no design de APIs, o que destoa com a realidade atual do cenário de desenvolvimento.

Em relação à linguagem GraphQL (será detalhada na Seção 4.2), segundo estudo realizado por Quiña-mera *et al*. (2023), onde foram analisados mais de 84 estudos relacionados a GraphQL com foco em tendências, lacunas de conhecimento e adoção do GraphQL como uma alternativa às APIs REST, há um interesse crescente pelas comunidades científica e industrial na linguagem GraphQL, com um aumento notável nas publicações e tendências de pesquisa. O estudo em questão ainda revela que a maioria das pesquisas é realizada em ambientes acadêmicos, mas com um número significativo de publicações em locais de alta qualidade. Também é enfatizado no estudo que a grande maioria das publicações são pesquisas de avaliação ou validação, com foco na comparação do GraphQL com REST e Simple Object Access Protocol (SOAP), sendo que muitas dessas publicações demonstraram vantagem de GraphQL em relação a termos de desempenho, tamanho da resposta e recursos de consulta dinâmica.

O framework gRPC (será detalhado na Seção 4.3) ainda está longe de alcançar os números da arquitetura REST, principalmente por ser uma tecnologia recente e ter um foco maior no cenário de computação distribuída, mas segundo Bronze, Farina e Florian (2023) em uma pesquisa realizada entre abril e agosto de 2023, a adoção da arquitetura gRPC por empresas de médio e grande porte é altamente viável devido à sua capacidade de melhorar a comunicação entre diferentes setores de uma empresa. De forma resumida, os autores destacaram que o gRPC é uma estrutura de comunicação robusta e que oferece uma abordagem moderna para a criação de sistemas distribuídos eficientes e dimensionáveis, com benefícios que superam suas desvantagens, tornando-o uma ferramenta valiosa para o setor de tecnologia.

Durante as pesquisas, poucos estudos foram encontrados com duas ou mais arquiteturas sendo comparadas e a grande maioria dos estudos encontrados são curtos e não se estendem ao nível técnico, somente ao nível conceitual. O estudo realizado por Buňata (2023) é bastante similar ao presente trabalho, onde foi realizado uma comparação entre REST, GraphQL e gRPC em uma API desenvolvida com Node.js. Porém, em quesitos de referências teóricas e principalmente em relação ao desenvolvimento das APIs e da realização dos testes, o estudo foi escasso e com escolhas que possivelmente afetaram os resultados apresentados pelo autor.

3.1. ANÁLISE

Todos os estudos analisados apresentaram e detalharam conceitos de um ou mais das arquiteturas que foram selecionadas e que auxiliaram no entendimento dos conceitos básicos para o desenvolvimento do trabalho, mas cada um com suas limitações e desafios próprios. Além disso, alguns dos estudos práticos realizaram experimentos com objetivos similares ao do presente trabalho, porém, apesar de alguns estudos serem mais completos em comparação com outros, todos possuem algum ponto negativo, como a falta de detalhamento das arquiteturas e tecnologias, falta de testes e código desenvolvido sem padrões de projeto que pode ter afetado os resultados. O presente estudo busca tratar e solucionar os pontos negativos apresentados acima.

4. ARQUITETURAS DE APIS

Esta seção busca detalhar a origem, características, aplicações, documentação, vantagens, desvantagens, dados de utilização e exemplos de cada arquitetura de desenvolvimento de API do presente estudo, além de realizar um breve comparativo entre as arquiteturas em estudo. Não foram abordadas todas as características e todos os conceitos de cada arquitetura, somente os tópicos principais de cada uma para comparação no presente estudo.

4.1. REST

4.1.1. ORIGEM

A arquitetura REST (Representational State Transfer ou Transferência de Estado Representacional) é um estilo arquitetônico para desenvolvimento de APIs que foi criado pelo cientista da computação norte-americano Roy Thomas Fielding (2000) durante sua tese de doutorado no ano de 2000. Segundo o próprio autor, esse estilo arquitetural foi elaborado para sistemas de hipermídia distribuídos, que nada mais são que sistemas de informação que permitem a interligação de diversos tipos de mídia, como texto, imagem, vídeo, áudio, seja através de links ou da própria navegação do sistema, ou seja, basicamente tudo que enxergamos, escutamos e interagimos na World Wide Web (WWW ou rede mundial de computadores) por meio da internet. Esses sistemas são os que atualmente dominam a internet, como sites, blogs, redes sociais e qualquer sistema que se enquadre nas características citadas anteriormente. O REST é um estilo arquitetural híbrido derivado de vários dos estilos arquitetônicos baseados em redes e combinado com restrições adicionais que definem uma interface de conector uniforme (FIELDING, 2000), sendo que tais restrições serão detalhadas na seção seguinte.

4.1.2. CARACTERÍSTICAS

Por se tratar de um modelo arquitetural, REST independe de uma linguagem de programação. Nesse sentido, é possível implementar uma API na arquitetura REST em diversas linguagens de programação. Segundo Fielding (2000), para que uma API seja desenvolvida na arquitetura REST é necessário que ela atenda às seguintes restrições:

- Estilo arquitetônico cliente-servidor: essa restrição remete ao conceito da separação de responsabilidades que rege o modelo cliente-servidor. Esse conceito detalha que ao separar as preocupações com a interface do usuário das preocupações com

o armazenamento de dados, melhoramos a portabilidade da interface do usuário em várias plataformas e melhoramos a escalabilidade simplificando os componentes do servidor. Nesse sentido, ainda é destacado que talvez o mais importante para a Web seja o fato de que a separação permite que os componentes evoluam de forma independente, dando suporte ao requisito de escala da internet de vários domínios organizacionais.

- Stateless (sem estado): essa restrição está totalmente relacionada com a restrição cliente-servidor, pois ela remete que a comunicação deve ser sem estado por natureza, de modo que cada solicitação do cliente para o servidor deve conter todas as informações necessárias para entender a solicitação e não pode tirar proveito de nenhum contexto armazenado no servidor. Portanto, o estado da sessão é mantido inteiramente no cliente. Essa restrição também induz as propriedades de visibilidade, confiabilidade e escalabilidade. A visibilidade é aprimorada porque um sistema de monitoramento não precisa olhar além de um único dado de solicitação para determinar a natureza completa da solicitação. A confiabilidade é aprimorada porque facilita a tarefa de recuperação de falhas parciais e a escalabilidade é aprimorada porque a não necessidade de armazenar o estado entre as solicitações permite que o componente do servidor libere recursos rapidamente e simplifica ainda mais a implementação porque o servidor não precisa gerenciar o uso de recursos entre as solicitações.
- Cache: na computação, um cache é uma camada de armazenamento de dados de alta velocidade que armazena um subconjunto de dados, geralmente de natureza transitória, para que futuras solicitações desses dados sejam atendidas mais rapidamente do que seria possível acessando o local de armazenamento principal dos dados. O cache permite que você reutilize com eficiência dados recuperados ou computados

anteriormente (AWS, 2023). Nesse sentido, complementando as restrições anteriores, as restrições de cache exigem que os dados em uma resposta a uma solicitação sejam implícitos ou explicitamente rotulados como armazenáveis em cache ou não armazenáveis em cache. Se uma resposta for armazenável em cache, um cache de cliente terá o direito de reutilizar esses dados de resposta para solicitações equivalentes posteriores.

- Interface uniforme: essa é a principal restrição que distingue o estilo arquitetônico REST de outros estilos baseados em rede, pois possui a ênfase em uma interface uniforme entre os componentes. Para obter uma interface uniforme, são necessárias várias restrições arquitetônicas para orientar o comportamento dos componentes. A REST é definida por quatro restrições de interface: identificação de recursos; manipulação de recursos por meio de representações; mensagens auto descritivas; e hipermídia como mecanismo de estado do aplicativo. Essas quatro restrições serão discutidas ao longo da Seção 3.1.
- Sistema em camadas: essa restrição permite que uma arquitetura seja composta de camadas hierárquicas, restringindo o comportamento do componente de modo que cada componente não possa "ver" além da camada imediata com a qual está interagindo. Ao restringir o conhecimento do sistema a uma única camada, é estabelecido um limite para a complexidade geral do sistema e promovido a independência do substrato, ou seja, em relação à infraestrutura subjacente.
- Code-on-demand (Código sob demanda): essa restrição é a única opcional no modelo arquitetural REST e permite que a funcionalidade do cliente seja ampliada por meio do download e da execução de código na forma de applets ou scripts, dessa forma isso simplifica os clientes ao reduzir o número de recursos que precisam ser pré-implementados. Por se tratar de uma

> restrição opcional e que não impacta no desenvolvimento de uma API na arquitetura REST, ela não será discutida com mais detalhes no presente trabalho.

Ainda segundo Fielding (2000), REST é uma abstração dos elementos arquitetônicos de um sistema de hipermídia distribuída. A arquitetura REST ignora os detalhes da implementação de componentes e da sintaxe do protocolo para se concentrar nas funções dos componentes, nas restrições de sua interação com outros componentes e na interpretação de elementos de dados significativos, abrangendo as restrições fundamentais sobre componentes, conectores e dados que definem a base da arquitetura da Web e, portanto, a essência de seu comportamento como um aplicativo baseado em rede. Nesse sentido, conforme descrito na Seção 3.1.2, a característica de interface uniforme da arquitetura REST possui algumas restrições que serão discutidas na sequência.

A ideia principal do modelo arquitetural REST é a abstração de informações em recursos que, segundo o autor, é qualquer informação que possa ser nomeada, como um documento, uma imagem, um serviço, uma coleção de outros recursos, um objeto, uma pessoa e assim por diante. Esses recursos são criados, acessados e modificados por meio de requisições e respostas, realizadas respectivamente pelo cliente e servidor. Para que um recurso possa ser identificado em uma comunicação via API REST, é preciso que cada recurso possua seu identificador, pois é o identificador do recurso que irá possibilitar a referência ao mesmo. A comunicação com REST é baseada no protocolo HTTP, que é o principal protocolo responsável pela transferência de dados na internet, nesse sentido, as requisições são executadas com base nos métodos HTTP, como POST, PUT, GET, DELETE, e demais métodos especificados no protocolo. As respostas do servidor podem ser retornadas em diversos formatos como JavaScript Object Notation (JSON), HTML, Extensible Markup Language (XML), texto, entre outros.

Exemplificando, uma API REST que foi desenvolvida em um modelo de negócios de um cinema, pode possuir funcionalidades que envolvem filmes, programação, estoque de alimentos, salas de cinema e ingressos. Um recurso possível seria o de ‘programação’, assim seria possível realizar ações como cadastrar uma nova programação semanal de filmes, modificar uma programação para incluir, atualizar ou remover filmes ou excluir uma programação que já se encerrou. Nesse sentido, ao cadastrar uma programação, essa programação iria receber um identificador (identificador do recurso) e para buscar ou manipular essa programação (recurso) esse identificador iria ser utilizado. Supondo que a programação de filmes da primeira semana do mês de fevereiro de 2024 tenha sido cadastrada e possui o identificador “020124”, para atualizar essa programação com um novo filme, poderia ser realizado uma requisição HTTP com o método PUT (indicado para atualização de dados) para o endereço da API no seguinte formato:

https://example-domain.com/api/schedule/020124

Sendo que ‘schedule’ seria referente ao recurso, que no caso do exemplo é o recurso ‘programação’ e na sequência o seu identificador ‘020124’ como parâmetro para que o recurso correto seja atualizado.

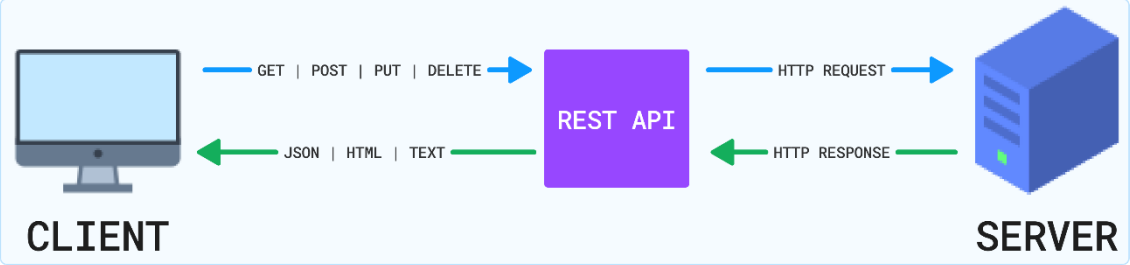


Figura 2 - Exemplificação da comunicação via API REST

Tendo em vista isso, o fluxo geral da comunicação via REST ocorre da seguinte forma (Fig. 2):

1. O cliente envia uma solicitação HTTP ao servidor com um método, recurso, parâmetro e tudo que for necessário de acordo com a documentação da API para formatar a solicitação de modo que o servidor entenda.
2. Caso a solicitação esteja de acordo com o que o servidor espera, a requisição é recebida e processada pelo servidor.
3. Por fim o servidor retorna uma resposta ao cliente com informações que indicam se a requisição foi realizada com sucesso ou não, e com as informações que podem ter sido solicitadas pelo cliente.

Os detalhes da solicitação e da resposta, como cabeçalhos, parâmetros, dados ou autenticação variam de acordo com a forma que a API foi desenvolvida.

A arquitetura REST não especifica nenhuma regra em relação à segurança, dessa forma fica a critério do desenvolvedor implementar meios de autorização/autenticação para deixar a API mais segura, seja por meio de código próprio, bibliotecas, softwares de terceiros, etc.

4.1.3. COMUNIDADE E DOCUMENTAÇÃO

Como citado anteriormente na Seção 3.1.1, o modelo arquitetural REST foi criado por Roy Thomas Fielding em sua tese de doutorado (FIELDING, 2000) no ano de 2000. Nesse sentido, a própria tese que está disponível na internet, serve como fonte de documentação do modelo arquitetural REST que foi proposto por ele. As APIs na arquitetura REST ainda prevalecem como as mais utilizadas na web atualmente, segundo pesquisa da empresa Postman (2023) realizada com 40.261 desenvolvedores e profissionais que trabalham com APIs no ano de 2023, e que é responsável por desenvolver uma das maiores plataformas para APIs com diversas ferramentas, a arquitetura REST é a mais utilizada para desenvolver APIs com 86% dos participantes informando que utilizam REST para desenvolver suas APIs nos projetos, nesse sentido, o

conteúdo referente a essa arquitetura é muito vasto na internet, com muitos artigos, blogs, vídeos e cursos que auxiliam a desenvolver uma API REST, além de plataformas da comunidade de desenvolvedores como a Stack Overflow (2024) onde diversas pessoas podem se ajudar com perguntas e respostas de assuntos diversos da área da computação.

4.1.4. VANTAGENS

A arquitetura REST para desenvolvimento de APIs possui diversas vantagens e, por isso, atualmente é a mais popular no cenário de desenvolvimento, conforme apresentado na seção anterior na pesquisa realizada pela empresa Postman (2023). Algumas das suas vantagens que são decorrentes das características, princípios e restrições apresentadas nas seções anteriores, são:

- Escalabilidade: as restrições como o modelo arquitetural cliente-servidor, o fato de não possuir estado, ter a possibilidade de utilizar cache e ser construída como um sistema em camadas, permite que a API possua uma ótima escalabilidade, que é o fator que permite que o sistema (API) possua a capacidade de manipular e manter suas funcionalidades mesmo com o crescimento das demandas, ou seja, está desenvolvida de forma que está preparada para um crescimento, seja no modelo de negócios ou em quantidades de processos.
- Flexibilidade: as APIs REST podem lidar com diversas solicitações e não são limitadas por nenhum formato específico, por se tratar de apenas um modelo arquitetural, REST pode ser utilizada para qualquer cenário, seja para realizar um login, integrar com outro sistema, gerar dados para relatórios, gerenciar soluções de pagamento, entre outras coisas, não há um limite e serve para todos os tipos de negócios.
- Independência dos componentes: devido a sua interface uniforme em um modelo cliente-servidor, é possível que vários clientes se

comuniquem simultaneamente com vários servidores, o que gera uma independência dos componentes e facilita o desenvolvimento de diversas coisas, o que também aumenta a flexibilidade e escalabilidade citadas anteriormente.

- Independência de formatos: também explicitado anteriormente, a arquitetura REST não exige nenhum formato específico de dado, seja para enviar ou receber informações, nesse caso diversos formatos podem ser utilizados para troca de informações, como JSON, HTML, XML, texto, entre outros, justamente pelo fato de utilizar o protocolo HTTP para executar as requisições e respostas.

4.1.5. DESVANTAGENS

Apesar de ser a arquitetura para desenvolvimento de APIs mais popular (POSTMAN, 2023) entre os desenvolvedores, REST também possui algumas desvantagens. Com base nas características, restrições e princípios apresentados anteriormente, é possível destacar certas desvantagens como:

- Falta de padronização: conforme apresentado na Seção 3.1.3, não existe uma documentação oficial para desenvolver uma API na arquitetura REST, apenas a tese do criador da arquitetura REST e conteúdos gratuitos ou pagos disponibilizados na internet, o que gera uma falta de padronização, pois os conteúdos disponibilizados na internet mesmo que se baseiem na dissertação de Roy Fielding podem possuir variações na implementação e interpretação dos princípios REST.
- Gestão de estado: por ser uma arquitetura sem estado, ou seja, na qual o estado da aplicação fica de responsabilidade do cliente no modelo cliente-servidor, conforme apresentado na Seção 3.1.2, em cenários com múltiplas interações e interações mais avançadas, a gestão do estado da aplicação pode se tornar mais complexa.

- Limitações: APIs desenvolvidas em REST podem enfrentar desafios com tarefas mais complexas, como tarefas que lidam com múltiplas transações ou atualizações atômicas.
- Versionamento: não existe na especificação REST um detalhamento a respeito de como lidar com novas versões de uma API, nesse sentido fica a critério do desenvolvedor escolher uma forma para que gerencia múltiplas versões de uma API REST.

4.2. GraphQL

4.2.1. ORIGEM

GraphQL é uma linguagem de consulta e um mecanismo de execução que foi originalmente criado pela empresa Facebook no ano de 2012 para descrever os recursos e os requisitos dos modelos de dados para aplicações no estilo cliente-servidor (GRAPHQL, 2024a). Em 2015 foi iniciado o processo para tornar a linguagem open standard, que significa padrão aberto em português, e a partir desse ano a linguagem passou a ser de livre acesso e implementação. No ano de 2019 foi criada a GraphQL Foundation para servir como um ponto focal neutro para organizações que apoiam o ecossistema GraphQL. Essa fundação foi hospedada pela Linux Foundation e tem como missão principal garantir que a comunidade GraphQL possa se concentrar na evolução contínua da especificação e das implementações de referência, por meio de atividades como financiamento de subsídios para mantenedores de projetos, coordenação e financiamento de programas de orientação, fornecimento de infraestrutura de colaboração, gerenciamento da marca registrada GraphQL e políticas relacionadas, fornecimento de suporte jurídico para projetos e suporte operacional e logística por meio da The Linux Foundation (GRAPHQL, 2024b).

4.2.2. CARACTERÍSTICAS

Conforme dito anteriormente, GraphQL é uma linguagem e um mecanismo de execução do lado do servidor, que possibilita a execução de consultas usando um sistema de tipos que é definido pelo próprio desenvolvedor,

ou seja, não se trata de um estilo arquitetural como REST, mas sim uma linguagem que possui sua especificação e sintaxe específica (GRAPHQL, 2024c). A linguagem GraphQL não é atrelada a nenhum banco de dados ou mecanismo de armazenamento específico, é apenas vinculada a seu código e dados existentes. Além disso, um serviço GraphQL pode ser desenvolvido com qualquer *framework backend* ou linguagem de programação (GRAPHQL, 2024c).

De forma resumida, a linguagem GraphQL trata basicamente da seleção de campos em objetos. Então, para que um serviço GraphQL seja criado, é preciso que seja definido um *schema*, que é um conjunto de tipos (objetos) que possuem campos (propriedades), e, associado a esse schema, também devem ser definidas funções para acesso a cada campo de cada tipo definido. Tendo em vista isso, existem três tipos desse conjunto que são especiais ou também conhecidos como *root* (raiz/base), o tipo "**Query**", o tipo "**Mutation**" e o tipo "**Subscription**", e servem respectivamente para as operações de consultas, mutações e assinaturas, tais operações serão descritas no próximo parágrafo. Além disso, o primeiro tipo (query) é obrigatório e os demais são opcionais para a execução de um serviço GraphQL.

Apesar de que em sua definição conste "linguagem de consulta", assim como uma API desenvolvida na arquitetura REST, uma API GraphQL, além de buscar dados também possibilita a manipulação e criação de dados. Tendo em vista isso, temos em GraphQL o tipo de operação, que pode ser "query" (consulta), "mutation" (mutação) ou "subscription" (assinatura), as quais servem, respectivamente, para uma busca somente de leitura, uma gravação seguida de uma busca e uma solicitação de longa duração que busca dados em resposta a eventos de origem (GRAPHQL, 2024a), sendo que esse último tipo não será discutido no presente trabalho devido à comparação entre as demais arquiteturas de APIs. Conforme parágrafo anterior, para que cada tipo de operação funcione, é necessário que seu tipo especial (objeto) também esteja definido no serviço GraphQL.

Utilizando o mesmo exemplo da Seção 3.1.2 para simular a criação de um serviço GraphQL, poderiam ser criados os tipos para uma programação

(*schedule*), para um filme e para uma consulta de uma programação, conforme exemplos apresentados nas figuras 3, 4 e 5.

```
type Movie {
  id: String // Identificador
  name: String // Nome
  description: String // Descrição
  release: String // Data de lançamento
  rating: String // Classificação
}
```

Figura 3 - Tipo e campos de um filme

```
type Schedule {
  id: String // Identificador da programação
  movies: Movie[] // Lista de filmes
  start_date: String // Data inicial
  end_date: String // Data final
}
```

Figura 4 - Tipo e campos de uma programação

```
type Query {
  schedule(id: String): Schedule // Programação
}
```

Figura 5 - Tipo especial Query e campo de uma consulta a uma programação pelo seu identificador

Após as definições de tipos, bastaria criar uma função para cada campo em cada tipo, como por exemplo, o tipo programação poderia ter as funções mostradas nas figuras 6, 7, 8 e 9.

```
function querySchedule(schedule_id: string) {
  // Lógica para buscar uma programação pelo identificador
  return schedule
}
```

Figura 6 - Função para buscar uma programação

```
function queryScheduleMovies(schedule_id: string) {
  // Lógica para buscar filmes de uma programação
  return schedule.movies
}
```

Figura 7 - Função para buscar filmes de uma programação

```
function queryScheduleStart(schedule_id: string) {
  // Lógica para buscar data inicial de uma programação
  return schedule.start_date
}
```

Figura 8 - Função para buscar a data inicial de uma programação

```
function queryScheduleEnd(schedule_id: string) {
  // Lógica para buscar data final de uma programação
  return schedule.end_date
}
```

Figura 9 - Função para buscar a data final de uma programação

Após o serviço GraphQL iniciar sua execução, que geralmente ocorre por meio de algum Uniform Resource Locator (URL), que é basicamente uma sequência de texto que especifica onde um recurso pode ser encontrado na Internet (MDN WEB DOCS, 2024b), ele pode receber consultas para validação e execução. Ao receber uma consulta, o serviço primeiramente verifica se a

consulta está solicitando tipos e campos que foram definidos e na sequência executa as funções fornecidas para gerar um resultado (GRAPHQL, 2024c). Assim como em uma API desenvolvida em REST é possível passar argumentos (parâmetros com valor fixo) ou variáveis (parâmetros com valor variável) para uma consulta. A diferença é que com GraphQL é possível passar um conjunto de argumentos ou variáveis para cada campo e objeto aninhado, sendo que em REST cada requisição recebe apenas um conjunto de argumentos ou variáveis.

Seguindo o exemplo da presente seção, uma possível consulta a uma programação específica seria conforme exemplificado nas figuras 10 e 11.

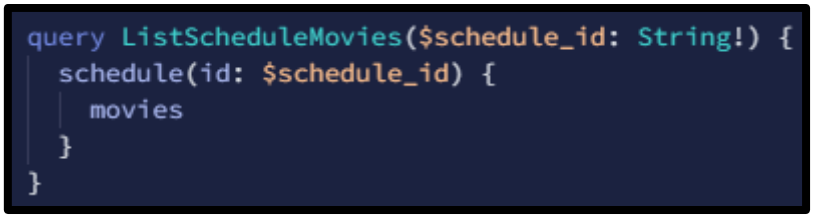


Figura 10 - Exemplo de consulta aos filmes da programação com id "020124" passado como argumento

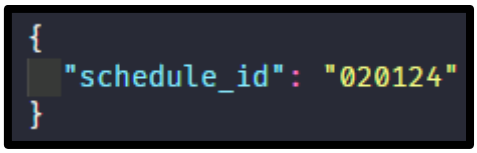
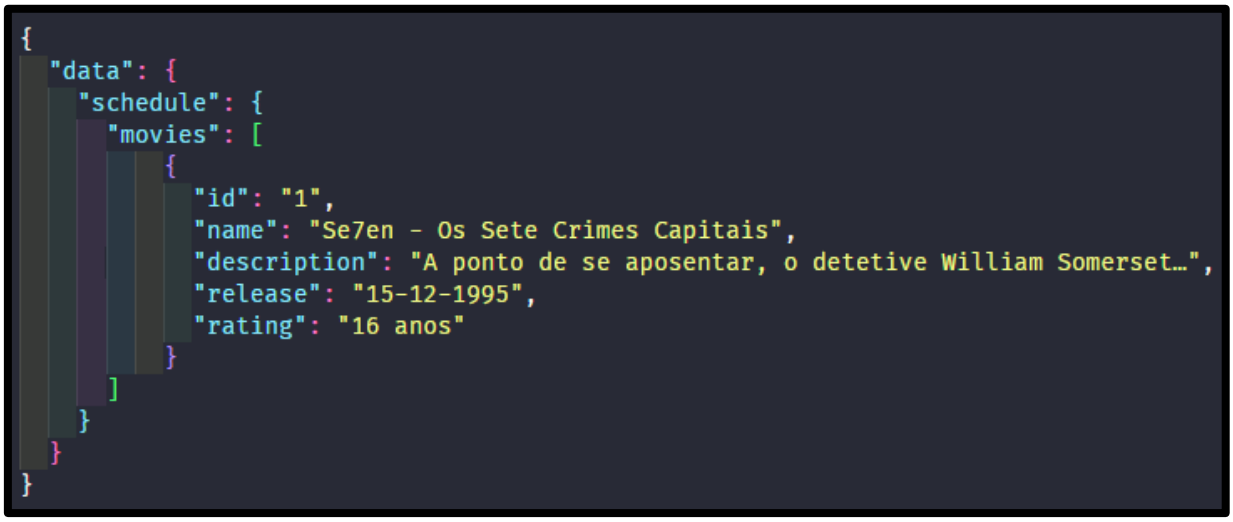


Figura 11 - Exemplo da variável informada na consulta

A consulta exemplificada na figura 10 poderia trazer o seguinte resultado (Fig. 12) em JSON.



```
{
  "data": {
    "schedule": {
      "movies": [
        {
          "id": "1",
          "name": "Se7en - Os Sete Crimes Capitais",
          "description": "A ponto de se aposentar, o detetive William Somerset…",
          "release": "15-12-1995",
          "rating": "16 anos"
        }
      ]
    }
  }
}
```


Figura 12 - Possível resultado em JSON da consulta de exemplo

Para o exemplo acima, temos que “**query**” é o tipo da operação, “**ListScheduleMovies**” o nome da operação, “**schedule**” o campo do tipo especial “**Query**” e **“$schedule_id**” a variável que remete ao identificador da programação seguindo o exemplo. O campo “**movies**” que também está presente nessa consulta será o campo que constará no resultado dessa operação.

Um exemplo de mutação seguindo o exemplo anterior, poderia ser a atualização de um filme, como a atualização de sua classificação, ficando da seguinte forma:

```
mutation UpdateMovieRating($movie_id: String, $rating: String) {
  updateMovieRating(movie_id: $movie_id, rating: $rating) {
    id
    name
    rating
  }
}
```

Figura 13 - Exemplo de mutação para atualizar a classificação de um filme

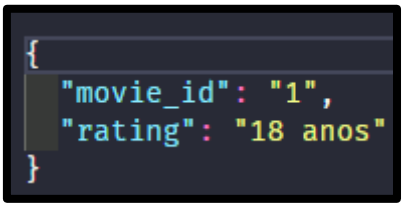


Figura 14 - Exemplo dos parâmetros informados na mutação

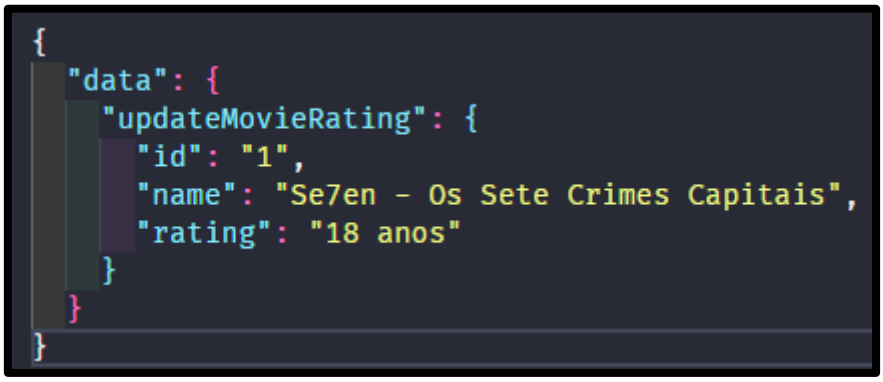


Figura 15 - Exemplo de retorno da mutação executada

Sendo “**mutation**” o tipo de operação, “**UpdateMovieRating**” o nome da operação, “**updateMovieRating**” o campo do tipo especial “**Mutation**” e **“$movie_id**” e **“$rating**” as variáveis que remetem respectivamente ao identificador do filme e ao novo valor da classificação do filme seguindo o exemplo. Os campos “**id**”, “**name**” e “**rating**” que também estão presentes nessa mutação serão os campos que constarão no resultado dessa operação. Um detalhe importante do exemplo anterior de mutação é que implicitamente deve existir um campo “**updateMovieRating**” no tipo especial (root) “**Mutation**” para que essa operação funcione.

Assim como na arquitetura REST, a linguagem GraphQL não possui uma regra determinada em relação aos aspectos de segurança, apenas uma recomendação de boa prática ao se aplicar métodos de autorização com a linguagem, dessa forma fica a critério do desenvolvedor aplicar camadas de segurança de autenticação/autorização por meio de código próprio, bibliotecas, softwares de terceiros, etc.

4.2.3. COMUNIDADE E DOCUMENTAÇÃO

Conforme mencionado na Seção 3.2.1, a linguagem GraphQL passou a ser de domínio público apenas no ano de 2015, sendo assim é uma tecnologia muito recente levando em consideração que REST teve sua origem no ano 2000. Além disso, apenas em 2019 foi criada a GraphQL Foundation que trouxe mais visibilidade para a tecnologia. Tendo em vista isso, a comunidade GraphQL ainda está em constante crescimento, mas já possui um grande número de adeptos. Segundo Postman (2023), a linguagem GraphQL ficou em terceiro lugar como arquitetura mais usada tendo o valor de 29% de uso pelos desenvolvedores que participaram da pesquisa, perdendo apenas para REST e Webhooks, mas ultrapassando a arquitetura SOAP (Simple Object Access Protocol) que é bastante difundida na Web e teve sua primeira versão lançada no ano de 1999. A principal fonte de documentação é o site oficial (https://graphql.org/) pois possui toda a especificação e documentação da tecnologia, além de informações da comunidade GraphQL com grupos da comunidade, da fundação GraphQL, perguntas frequentes, blog, cursos, exemplos e diversas outras informações.

Em uma API GraphQL, da mesma forma que em uma API REST, as operações ocorrem sobre o protocolo HTTP, porém não existe nenhuma regra na especificação da linguagem que obrigue que as operações sejam executadas de uma forma específica. Existe apenas a recomendação de boa prática para utilizar o método HTTP GET para operações de consulta e o método HTTP POST para operações de mutação (GRAPHQL, 2024c), diferentemente da arquitetura REST onde dependendo da operação recomenda-se um método HTTP específico para utilização, como GET para buscas, POST para cadastros, PUT para atualizações e DELETE para exclusões.

4.2.4. VANTAGENS

As características específicas que podem projetar alguma vantagem na abordagem de GraphQL em relação a arquitetura REST, de acordo com estudo realizado por Quiña-mera *et al*. (2023) são:

- Previsibilidade: os resultados obtidos em operações GraphQL são mais previsíveis pois fica a cargo do cliente especificar quais dados deseja receber como resposta.
- Desempenho: estudos mostram que em termos de desempenho, o GraphQL pode ser mais rápido que o REST em consultas simples de um único endpoint. Em consultas complexas, o GraphQL pode ser até 187 vezes mais rápido que o REST.
- Tamanho da resposta: as consultas com GraphQL tendem a ter um tamanho de resposta menor em relação a REST, o que pode resultar em uma comunicação mais eficiente entre cliente e servidor.
- Menos solicitações: diferentemente de REST onde para obter múltiplos recursos são necessárias múltiplas requisições, com uma API GraphQL espera-se que seja possível obter múltiplos recursos com apenas uma única consulta, o que consequentemente pode gerar uma quantidade menor de consultas ao servidor.
- Tipagem forte: para construir um serviço com GraphQL é necessário a definição de tipos e campos, diferentemente de REST na qual a API é construída e acessada por meio de recursos. Com tipos e campos definidos espera-se que uma API GraphQL garanta que os clientes solicitem apenas o que é possível e além disso que forneça erros mais claros e úteis.
- Interoperabilidade: com GraphQL há um aumento na reutilização de operações, reduzindo a necessidade de coordenação entre desenvolvedores frontend e backend. Além disso, oferece uma sintaxe melhor para leitura de código e menos esforço na especificação de parâmetros.
- Versionamento: com GraphQL, é possível adicionar novos campos e tipos a consultas já existentes sem afetá-las, permitindo uma única versão com acesso contínuo, o que consequentemente pode gerar um código mais limpo e de fácil manutenção do lado do servidor.

4.2.5. DESVANTAGENS

Apesar de que GraphQL possui diversas vantagens para a construção de uma API, também possui certas desvantagens, segundo estudo realizado por Quinã-mera *et al*. (2023), podemos citar:

- Campos privados: no conjunto de tipos coletivos (schemas) de aplicações GraphQL, não há suporte para campos privados, o que permite uma visualização de campos desse tipo para as aplicações cliente que possuam acesso.
- Grandes conjuntos de tipos: em aplicações GraphQL com grandes conjuntos de tipos, há uma maior complexidade para a compreensibilidade e implementação de consultas complexas, além de também ter uma maior probabilidade de colisões em nome de objetos.
- Cache: a linguagem GraphQL não realiza o controle de cache conforme a especificação HTTP, pois utiliza apenas um único endpoint (endereço de conexão da API para receber as chamadas). Essa limitação pode ser corrigida com a implementação de uma biblioteca de gerenciamento de cache.

4.3 gRPC

4.3.1 ORIGEM

Segundo o site oficial, o gRPC (gRPC Remote Procedure Calls) foi criado inicialmente pela empresa Google, que há mais de uma década utilizava uma única infraestrutura de RPC de uso geral chamada Stubby para conectar o grande número de microsserviços executados dentro e entre seus data centers (GRPC, 2024b). A partir disso, em março de 2015, a Google decidiu criar uma nova versão do Stubby e torná-la de código aberto, o que resultou na criação do gRPC. Apesar de ter sido criada para utilização interna da empresa, o gRPC pode ser de código aberto já é utilizado por diversas organizações como Netflix, Square, Cisco, entre outras (GRPC, 2024b). Ainda segundo o site oficial, o gRPC é conceituado como uma estrutura moderna de RPC (Remote Procedure Call ou Chamada de Procedimento Remoto) de código aberto e que pode ser executada em qualquer ambiente, servindo para diversos casos de uso, desde a conexão entre serviços como também na última milha da computação, servindo para conectar dispositivos, aplicativos móveis e navegadores a serviços back-end (GRPC, 2024a).

4.3.1 CARACTERÍSTICAS

Como consta no próprio nome, a implementação do gRPC é baseada em RPC, que é um protocolo que fornece o paradigma de comunicações de alto nível usado no sistema operacional (IBM, 2024). O protocolo RPC pressupõe a existência de um protocolo de transporte de baixo nível, como o Protocolo de Controle de Transmissão/Protocolo de Internet (TCP/IP) ou o Protocolo de Datagrama do Usuário (UDP), sendo o último o mais utilizado geralmente, para transportar os dados da mensagem entre os programas de comunicação. O RPC implementa um sistema lógico de comunicação cliente-servidor projetado especificamente para o suporte de aplicativos de rede (IBM, 2024). Nesse sentido, com gRPC, uma aplicação cliente pode chamar um método diretamente em uma aplicação servidor em uma máquina diferente, como se fosse um objeto local, o que consequentemente facilita a criação de aplicativos e serviços distribuídos (GRPC, 2024c). Assim como em muitos sistemas que utilizam RPC, o gRPC baseia-se na ideia de definir um serviço, especificando os métodos que podem ser chamados remotamente com seus parâmetros e tipos de retorno. No lado do servidor, o servidor implementa essa interface e executa um servidor gRPC para lidar com as chamadas do cliente. No lado do cliente, o cliente tem um *stub* (chamado apenas de cliente em algumas linguagens) que fornece os mesmos métodos que o servidor (GRPC, 2024c).

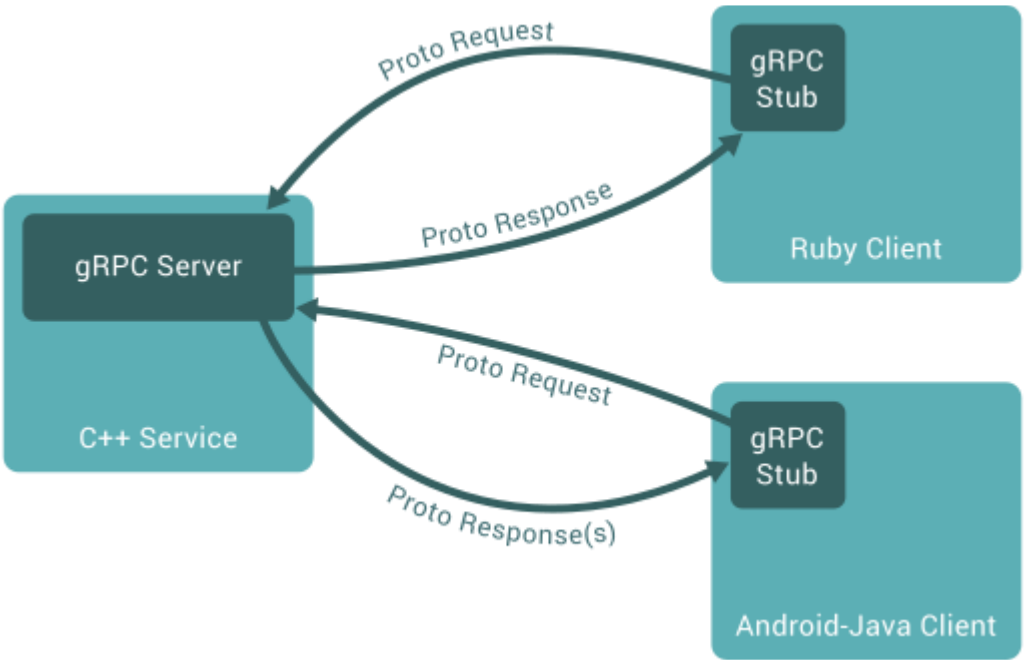


Figura 16 - Exemplo de comunicação utilizando gRPC

Ainda segundo o site oficial (GRPC, 2024c), os clientes e servidores gRPC podem ser executados e se comunicarem em vários tipos de ambientes, além de ter suporte a diversas linguagens. Dessa maneira, conforme exemplo da Figura 16, é possível criar um servidor gRPC utilizando uma linguagem como C++ e utilizar clientes com outras linguagens como Ruby ou Java para consumir o serviço normalmente.

Por padrão, um servidor gRPC utiliza ‘Protocol Buffers’ (também conhecido como protobuf ou em português buffers de protocolo) como linguagem de definição de interface, que é uma linguagem para serialização de dados estruturados de código aberto que também foi criado pela Google, mas que não impede que possa ser utilizado outros formatos de dados como JSON (GRPC, 2024c). A seção seguinte irá detalhar o funcionamento dos protocol buffers.

Para ser possível a comunicação com buffers de protocolo, é necessário primeiramente definir a estrutura dos dados que será serializada em um arquivo proto, que é basicamente um arquivo de texto comum, mas que possui a extensão '.proto'. Os dados de um buffer de protocolo são estruturados como mensagens, sendo que cada mensagem é um pequeno registro lógico de informações que contém uma série de pares nome-valor que são chamados de campos (GRPC, 2024c). Conforme a figura 17, temos como exemplo uma mensagem chamada **'Person'** que possui os campos **name**, **id** e **has_job** com seus respectivos tipos e números identificadores. Segundo documentação oficial da linguagem 'Protocol Buffers' (LANGUAGE…, 2024), os números identificadores são obrigatórios e únicos em cada mensagem e servem para o processo de codificação dos dados.

```
message Person {
  string name = 1;
  int32 id = 2;
  bool has_job = 3;
}
```

Figura 17 - Exemplo de uma mensagem definida em um arquivo proto

Também em arquivo *proto*, é necessário definir os serviços gRPC com seus respectivos parâmetros de método RPC e seus tipos de parâmetros de entrada e de retorno que devem ser especificados como mensagens de protocol buffer, conforme exemplo da figura anterior. Na Figura 18 temos como exemplo a definição de um serviço gRPC simples que realiza uma saudação (***service Greeter***), um método RPC nesse serviço chamado '***SayHello'*** que recebe como parâmetro uma mensagem do tipo **'*HelloRequest'*** que contém um campo **'nome'** e retorna uma mensagem do tipo **'*HelloReply'*** que contém um campo **'mensagem'**.

```
// A definição do serviço de exemplo Saudação
service Greeter {
  // Método RPC que envia uma saudação
  rpc SayHello (HelloRequest) returns (HelloReply) {}
}

// A mensagem da requisição contendo o nome do usuário
message HelloRequest {
  string name = 1;
}

// A mensagem de resposta contendo a saudação
message HelloReply {
  string message = 1;
}
```

Figura 18 - Exemplo de um serviço gRPC definido em um arquivo proto

Em relação aos métodos RPCs de cada serviço gRPC, segundo a documentação oficial (GRPC, 2024d), é possível criar quatro tipos:

- RPCs unários no qual o cliente envia uma única solicitação ao servidor e recebe uma única resposta.
- RPCs de streaming (fluxo contínuo) de servidor no qual o cliente envia uma solicitação ao servidor e recebe um retorno de um fluxo de mensagens em sequência do servidor.
- RPCs de streaming (fluxo contínuo) de cliente no qual o cliente envia um fluxo de mensagens em sequência para o servidor e recebe uma única resposta.
- RPCs de streaming (fluxo contínuo) bidirecional no qual tanto cliente como servidor enviam um fluxo de mensagens em sequência.

Com a definição de um serviço gRPC em um arquivo *proto*, o gRPC por meio do *Protocol Buffer Compiler Installation*, fornece plugins de compilador de buffer

de protocolo que geram código tanto do lado do cliente como do lado do servidor. Após isso, a utilização desse código gerado é utilizado para implementar a API correspondente no lado do servidor e consequentemente para que a API do servidor seja utilizada pelo lado do cliente. Dessa forma, o servidor implementa os métodos do serviço que foram declarados no arquivo proto e executa um servidor gRPC para responder às chamadas do cliente. A infraestrutura gRPC decodifica as solicitações recebidas, executa os métodos do serviço e codifica as respostas do serviço. No lado do cliente há um objeto local conhecido como stub (em certas linguagens o termo preferido é cliente) que implementa os mesmos métodos do serviço, dessa forma, o cliente pode chamar os métodos no objeto local que envolvem os parâmetros da chamada no tipo de mensagem de buffer de protocolo correspondente, envia essas solicitações ao servidor e retorna as respostas de buffer de protocolo do servidor (GRPC, 2024d).

Apesar de cada API possuir sua própria forma de autenticação/autorização que vai depender da escolha do desenvolvedor, o framework gRPC possui nativamente os seguintes recursos (GRPC, 2024e):

- SSL/TLS: possui integração com SSL/TLS e promove o uso de SSL/TLS para autenticar o servidor e criptografar todos os dados trocados entre o cliente e o servidor, além disso, há mecanismos opcionais que estão disponíveis para que os clientes forneçam certificados para autenticação mútua.
- ALTS: caso o serviço gRPC esteja sendo executado no Google Kubernetes Engine (GKE) ou no Compute Engine, há suporte ao ALTS como um mecanismo de segurança de transporte.
- Autenticação baseada em token com o Google: o gRPC fornece um mecanismo genérico para anexar credenciais baseadas em metadados a solicitações e respostas.

4.3.2 COMUNIDADE E DOCUMENTAÇÃO

O framework gRPC se tornou de código aberto apenas no ano de 2015, e, por esse motivo, é uma tecnologia muito recente e com pouco tempo de

mercado se comparada com o modelo arquitetural REST que teve sua origem no ano de 2000, entretanto foi lançada no mesmo ano que a linguagem GraphQL. Apesar de ser uma tecnologia recente, segundo o relatório Postman (2023), gRPC ficou em sexto lugar como arquitetura mais utilizada, com pouco mais de 10% de uso pelos desenvolvedores que participaram da pesquisa. No site oficial da tecnologia, existe uma seção destinada a informações da comunidade gRPC, com canais de comunicação oficiais como Twitter, Google Groups, Gitter, Stack Overflow, Reddit, YouTube, entre outros, o que demonstra que existe uma preocupação com a comunidade que utiliza a tecnologia (GRPC, 2024e). Também são organizados encontros mensais com a comunidade para discutir ideias e solucionar problemas, além de promover conferências. O site oficial também conta com uma documentação geral sobre os principais conceitos do *framework* e seções específicas com guias para cada linguagem/plataforma que possui suporte a gRPC, o que facilita para aprender sobre a tecnologia.

4.3.2 VANTAGENS

Apesar de inicialmente ter sido originado para comunicação de serviços entre data centers, atualmente o framework gRPC possui diversos cenários de uso e algumas de suas vantagens são:

- Multiplataforma: gRPC possui suporte a plataformas como Web, Android e em breve terá suporte a iOS e Flutter (GRPC, 2024f), ou seja, permite a conexão de dispositivos móveis e também navegadores a serviços backend.
- Multilinguagem: além de ser multiplataforma também possui suporte a 11 linguagens atualmente, sendo elas C#/.NET, C++, Dart, Go, Java, Kotlin, Node, Objective-C, PHP, Python e Ruby (GRPC, 2024f).
- Transporte HTTP: a arquitetura possui seu transporte baseado no protocolo HTTP/2 que traz diferenciais significativos em relação ao HTTP/1, como a possibilidade de transmissão bi-direcional (GRPC, 2024b).
- Buffer de protocolo: por utilizar buffer de protocolo para a serialização dos dados espera-se uma maior eficiência na transferência de dados com tamanho menor de mensagens.

- Tipagem forte: assim como em GraphQL, um serviço gRPC depende de uma pré-configuração das mensagens e métodos com seus tipos definidos por meio dos arquivos proto.
- Também possui outras funcionalidades como suporte a autenticação, rastreamento, balanceamento de carga e verificação de integridade (GRPC, 2024f).

4.3.3 DESVANTAGENS

Apesar de ter diversas vantagens conforme explicitado na seção anterior, as principais desvantagens ao considerar a criação de um serviço com gRPC são:

- Complexidade: a implementação de um serviço gRPC exige o conhecimento dos buffers de protocolo e suas particularidades, além de demandar mais tempo no momento do desenvolvimento se comparada com uma API REST que é possível criar com apenas um arquivo de código.
- Depuração: também devido a sua natureza binária com os buffers de protocolo, é mais difícil depurar um serviço gRPC do que um serviço REST.

4.4 COMPARATIVO

Conforme seções anteriores, todas as arquiteturas de API do presente estudo possuem suas particularidades e seus prós e contras, seja de implementação, configuração ou utilização. Com base nas informações apresentadas na Seção 4, temos na tabela 3 as principais características das arquiteturas com suas diferenças.

Tabela 3 - Comparativo entre a arquitetura REST, GraphQL e gRPC

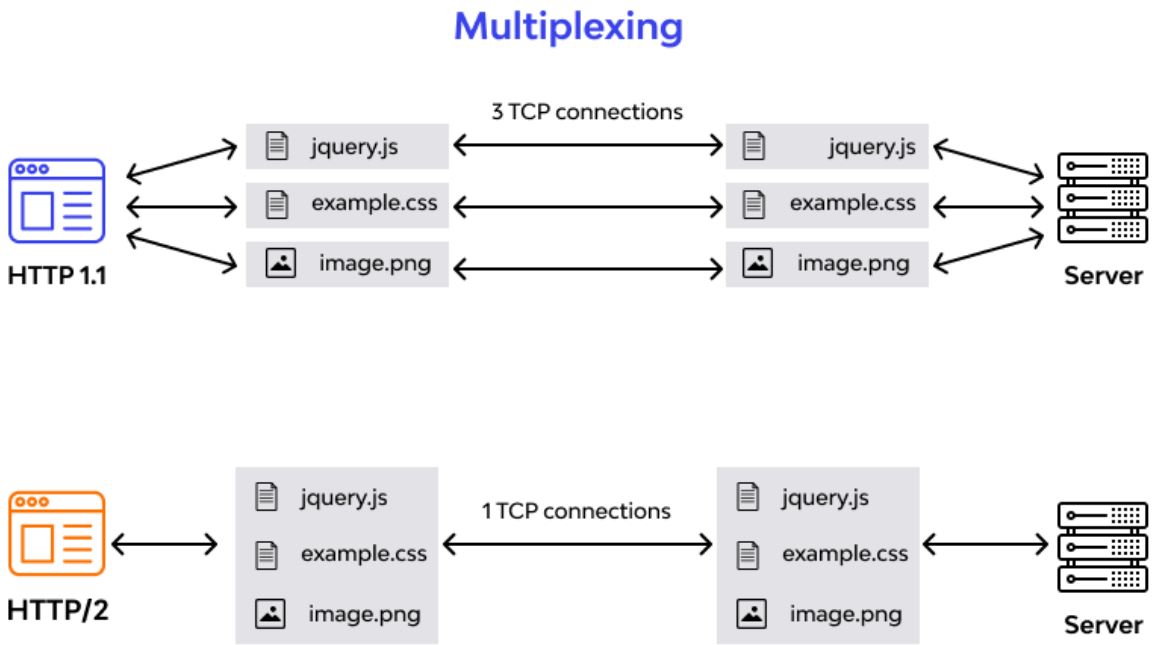
Arquitetura	**REST**	**GraphQL**	**gRPC**
Protocolo HTTP	HTTP/1.1	HTTP/1.1	HTTP/2
Formato das mensagens	JSON (mais comum), XML, texto ou outros	JSON (mais comum) ou outros	Buffers de protocolo (mais comum), JSON ou outros
Comunicação	Solicitação de cliente	Solicitação de cliente	Solicitação de cliente ou fluxo contínuo (servidor, cliente ou bidirecional)
Implementação	Rápida	Rápida	Lenta
Abordagem	Orientado a entidades (recursos)	Orientado a consultas	Orientado a serviços
Geração de código	Requer ferramentas de terceiros	Recurso integrado	Recurso integrado
Tratamento de erros (entrada e validação de dados)	Fraco - responsabilidade do cliente	Médio - possui tratamento integrado para certos erros	Médio - possui tratamento integrado para certos erros, mas sem suporte a todas as linguagens/plataformas
Suporte a autenticação/autorização	Não possui regra, pode ser implementada pelo desenvolvedor por meio de bibliotecas, softwares, etc.	Não possui regra, pode ser implementada pelo desenvolvedor por meio de bibliotecas, softwares, etc.	Integrado com SSL/TLS, ALTS e autenticação baseada em token com Google. Também podem ser implementados de outras formas pelo desenvolvedor por meio de bibliotecas, softwares, etc.
Tipagem	Fraca	Forte	Forte

Uma das principais diferenças é a versão do protocolo HTTP utilizada para a comunicação com a API. O protocolo HTTP ou protocolo de transferência de hipertexto serve para buscar recursos, como por exemplo um documento HTML, e é a base de qualquer troca de dados na Web (MDN WEB DOCS, 2024a). Além disso é um protocolo cliente-servidor, o que significa que as

solicitações são iniciadas pelo destinatário, que no cenário da Web geralmente é um navegador. Conforme a tabela anterior, tanto REST como GraphQL fazem uso da versão 1.1 do protocolo HTTP, que foi a primeira versão utilizável do protocolo lançada no ano de 1997. Já as APIs desenvolvidas com gRPC utilizam a versão 2 do protocolo HTTP que foi lançada oficialmente no ano de 2015 (MDN WEB DOCS, 2024a). Os grandes diferenciais da versão 2 se comparada a anterior são (MDN WEB DOCS, 2024a):

- É um protocolo binário em vez de um protocolo de texto. Ele não pode ser lido e criado manualmente. Apesar desse obstáculo, ele permite a implementação de técnicas de otimização aprimoradas.
- É um protocolo multiplexado. Solicitações paralelas podem ser feitas na mesma conexão, removendo as restrições do protocolo HTTP/1.x.
- Ele comprime os cabeçalhos. Como eles geralmente são semelhantes em um conjunto de solicitações, isso elimina a duplicação e a sobrecarga dos dados transmitidos.
- Permite que um servidor preencha os dados em um cache de cliente por meio de um mecanismo chamado push de servidor.

Figura 19 - Exemplo do funcionamento de 'multiplexing' em requisições HTTP



Fonte: Wallarm, 2024

Todas essas características da versão 2 do protocolo HTTP trazem melhorias significativas em comunicações com APIs, como, por exemplo, a possibilidade de solicitações de *streaming* (fluxo contínuo) do cliente para o servidor ou vice-versa, que é possível com uma API gRPC por exemplo.

Com base no estudo de cada tecnologia, é notável que cada uma possui casos de uso mais adequados em relação às demais, tendo influência ou não de sua origem. Por se tratar de uma arquitetura, é possível aplicar REST em qualquer cenário, já GraphQL pode trazer um maior controle do cliente nas solicitações realizadas, pode funcionar melhor em cenários onde existem consultas mais complexas ou diversificadas. O framework gRPC foi concebido inicialmente para comunicação entre serviços de um data center e também entre outros data centers, o que faz dele uma ótima opção para cenários como esses e também cenários de sistemas distribuídos e microsserviços.

Apesar das características de cada uma, atualmente existem diversos frameworks, bibliotecas e softwares de terceiros que auxiliam na implementação de uma API, seja em REST, GraphQL ou gRPC. Como exemplo, no desenvolvimento do presente estudo foi utilizado o framework NestJS que possui suporte a todas as arquiteturas mencionadas anteriormente e será detalhado na seção 5.3.

5 DESENVOLVIMENTO

5.1 PROPOSTA

Apesar de já existirem estudos que demonstram o desempenho entre as arquiteturas de APIs REST, GraphQL e gRPC, nem todos os estudos podem ser utilizados inteiramente como fonte de informação e escolha para uma tomada de decisão, visto que na área de desenvolvimento de software existem muitos fatores que podem influenciar o desempenho de uma aplicação, como a forma que ela foi desenvolvida, a conexão com a internet, a performance do computador onde a aplicação está hospedada, a linguagem de programação utilizada, e, a forma que foi implementada. Dessa forma, a solução proposta do presente trabalho é gerar mais um estudo para a literatura, porém com a aplicação de boas práticas de desenvolvimento e uma boa metodologia, para que possa ser útil e de confiabilidade para a comunidade de desenvolvimento de software. O principal objetivo é realizar um estudo das arquiteturas de APIs REST, GraphQL e gRPC e desenvolver uma API que contemple as mesmas, que permita a discussão e análise das características, limitações, pontos positivos e negativos e o desempenho geral de cada uma utilizando a ferramenta para testes k6 da empresa Grafana para obter as métricas gerais de quantidade de dados recebidos, quantidade de dados enviados e tempo para concluir uma iteração completa, além de métricas específicas para o protocolo HTTP, sendo elas, tempo total de uma requisição, taxa de solicitações com falha, tempo gasto para estabelecer a conexão TCP, tempo gasto no envio de dados, tempo gasto para receber dados de resposta e total de solicitações HTTP, bem como métricas específicas para a arquitetura gRPC, com o tempo para receber uma resposta,

número total de fluxos iniciados, número total de mensagens recebidas e número total de mensagens enviadas. Como resultado, espera-se que o estudo sirva como fonte de informação para desenvolvedores e arquitetos que buscam desenvolver novas APIs e estejam em dúvida de qual selecionar para o caso de uso de sua necessidade. Nesse sentido, a solução final será uma API completa que terá a implementação das três arquiteturas de API citadas anteriormente, além da realização de testes nessa API para conclusão do estudo e geração de novas hipóteses.

5.2 MODELAGEM

A solução proposta será o desenvolvimento de uma API que permita a comunicação via requisições HTTP com arquitetura REST, GraphQL e gRPC. A API irá utilizar o banco de dados com dados iniciais de filmes e avaliações contidas no conjunto de dados ‘Letterboxd’ (GARANIN, 2024) que contém dados de mais de 850 mil filmes, com descrições, atores, equipes e lançamentos, pois na etapa de testes o comparativo entre as arquiteturas será feito com uma quantidade de dados pequena e também com uma quantidade de dados grande para análise de desempenho. Essa API irá permitir operações com filmes, atores dos filmes e equipes dos filmes, e contará com os seguintes casos de uso:

- Cadastro de um ator;
- Edição de dados de um ator;
- Exclusão de um ator;
- Listagem de atores;
- Cadastro de um novo filme;
- Edição de dados de um filme;
- Exclusão de um filme;
- Listagem de filmes;
- Cadastro de uma equipe de um filme;
- Edição de dados de uma equipe de um filme;
- Exclusão de uma equipe de um filme;
- Listagem da equipe de um filme;

A solução contará com os seguintes requisitos não funcionais:

- A API deve ser desenvolvida em um ambiente com sistema operacional Windows 10 ou superior.
- A arquitetura da API deve ser desenvolvida em camadas, com um baixo acoplamento permitindo que um mesmo caso de uso seja executado via REST, GraphQL ou gRPC.

5.3 FERRAMENTAS

As seguintes ferramentas foram utilizadas para o desenvolvimento, utilização e teste da solução do presente estudo:

5.3.1 Node.js

O Node.js é um ambiente de tempo de execução JavaScript de código aberto e multiplataforma, o qual permite que desenvolvedores criem aplicações JavaScript para rodar como uma aplicação standalone (programa autossuficiente que não necessita de um software auxiliar) em uma máquina, não dependendo de um browser para a execução (NODE.JS, 2024), ou seja, é a ferramenta que irá permitir a execução de código JavaScript no lado do servidor, o qual será o foco do desenvolvimento da solução.

5.3.2 NestJS

NestJS é um framework de Node.JS que segundo site oficial permite a criação de aplicações eficientes, confiáveis e escaláveis no lado do servidor (NESTJS, 2024). Ainda segundo o site oficial, o framework oferece uma grande flexibilidade pois permite o uso de qualquer outra biblioteca devido a sua arquitetura modular. Além disso é um ecossistema adaptável que permite a ligação entre todos os tipos de aplicativos do lado do servidor e também

aproveita os recursos mais recentes do JavaScript, levando padrões de design e soluções mais maduras para o mundo do Node.JS (NESTJS, 2024). Esse framework foi escolhido para o desenvolvimento da solução devido a sua grande versatilidade para integrar com diversas bibliotecas (algumas também fornecidas pelo framework) as quais facilitaram o desenvolvimento do presente estudo, além de uma ampla e completa documentação.

5.3.3 TypeScript

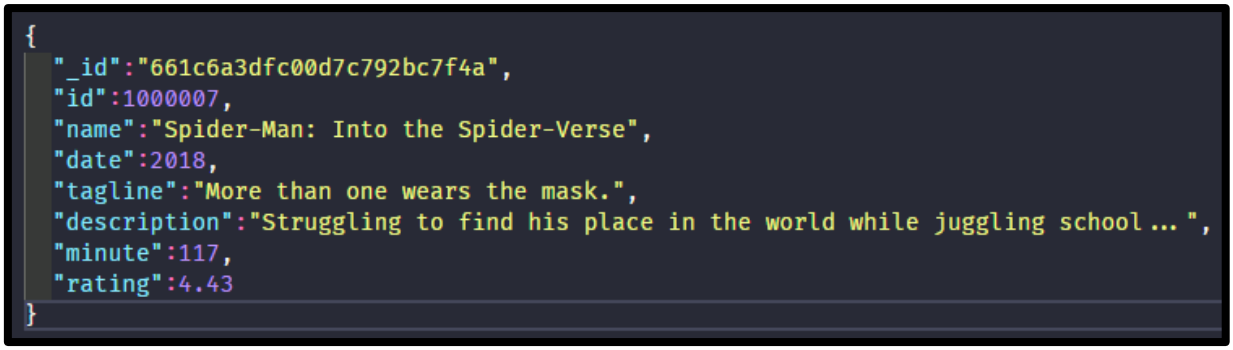
Segundo o site oficial, o TypeScript é uma linguagem de programação fortemente tipada que se baseia no JavaScript, ou seja, de forma resumida é uma melhoria da linguagem de programação JavaScript com a adição da sintaxe para tipos (TYPESCRIPT, 2024). O TypeScript além de acrescentar sintaxe adicional ao JavaScript, também oferece suporte a uma integração mais estreita com os editores de código, permitindo que erros sejam detectados antecipadamente. O código desenvolvido em TypeScript é convertido em JavaScript que então pode ser executado em qualquer lugar onde o JavaScript possa ser executado, como nos navegadores ou em uma aplicação Node.JS (TYPESCRIPT, 2024).

5.3.4 MongoDB

O banco de dados utilizado na solução desenvolvida é o MongoDB, que segundo Mongodb (2024) é um banco de dados de documentos projetado para facilitar o desenvolvimento e o dimensionamento de aplicativos. O MongoDB é um banco de dados não relacional, também conhecido como “NoSQL”, que diferentemente de um banco de dados relacional que trabalha com linhas e tabelas, trabalha com coleções que armazenam documentos. Qualquer registro no banco de dados MongoDB é um documento, que é uma estrutura de dados composta de pares de campos e valores, bem semelhante aos objetos JSON. Esses campos podem incluir outros documentos, listas e listas de documentos (MONGODB, 2024). Em um banco de dados relacional é necessário um

esquema pré-definido antes de inserir dados, já em um banco de dados não relacional como MongoDB isso não é necessário (ORACLE, 2024).

Exemplificando, para realizar o cadastro de um filme em um banco de dados MongoDB, bastaria a criação de um documento com as informações necessárias de um filme sem definir um esquema, logo o seguinte JSON seria válido para uma inserção no banco de dados:



```
{
  "_id":"661c6a3dfc00d7c792bc7f4a",
  "id":1000007,
  "name":"Spider-Man: Into the Spider-Verse",
  "date":2018,
  "tagline":"More than one wears the mask.",
  "description":"Struggling to find his place in the world while juggling school ... ",
  "minute":117,
  "rating":4.43
}
```


Figura 20 - Exemplo de formato aceito pelo banco de dados MongoDB

Em contrapartida, para inserir esse filme em um banco de dados relacional, seria necessário a criação de uma tabela “Filmes” com a definição de todos os campos com seus respectivos tipos e referências, para então conseguir inserir o filme no formato adequado de sua tabela.

5.3.5 Mongoose

Em conjunto com o banco de dados MongoDB, a solução utiliza a biblioteca Mongoose, que é um ODM (Object Document Mapping) e serve para fazer a modelagem de objetos MongoDB para Node.js, ou seja, faz uma integração com o banco de dados e o Node.js trazendo diversas facilidades para o desenvolvimento como conversão de tipos incorporada, validação, criação de consultas e diversas outras funcionalidades que já são prontas para uso (MONGOOSE, 2024). Conforme explicitado, essa ferramenta possui diversas funcionalidades e uma documentação extensa que pode ser acessada por meio de seu site principal (MONGOOSE, 2024).

5.3.6 Insomnia

Como a solução não teve uma parte visual (*front-end*) desenvolvida, para testar a API foi necessário a utilização de uma ferramenta que permitisse testes em APIs. Para isso foi utilizado a ferramenta Insomnia, a qual permite projetar, depurar, testar e simular APIs, seja localmente ou até mesmo na nuvem (INSOMNIA, 2024a). Possui diversos plugins, suporte a diversos protocolos (incluindo todos do presente estudo), permite testes de uma maneira fácil e organizada (INSOMNIA, 2024a), além de diversas outras funcionalidades que estão documentadas em seu site de documentação (INSOMNIA, 2024b).

5.3.7 Grafana k6

Para realizar os cenários de teste da solução desenvolvida no estudo, foi utilizada a ferramenta k6 da empresa Grafana Labs, que é uma ferramenta de código aberto para testes de carga, além de também permitir testes end-to-end para Web, testes de infraestrutura, testes de injeção de falhas, monitoramento sintético e testes de regressão (K6, 2024). Possui também diversas integrações e extensões, e com scripts implementados em javascript, é possível realizar os testes de carga localmente, de forma distribuída ou na nuvem (K6, 2024). É uma ferramenta completa utilizada por grandes empresas como Amazon e Microsoft e que possui diversos recursos com uma documentação extensa que abrange configurações, utilização e guias de teste que auxiliam desde cenários mais básicos até cenários mais complexos utilizando a ferramenta. Com a ferramenta foi possível criar scripts de teste para o acesso a API com REST, GraphQL e gRPC. É possível executar testes simulando a utilização de múltiplos usuários virtuais e por um certo período de tempo, entre outras funcionalidades.

5.4 RESULTADOS

Conforme explicitado na Seção 5.2, a API foi desenvolvida com uma arquitetura que permite que um mesmo caso de uso seja executado via REST,

GraphQL ou gRPC, ou seja, o que diferencia é a porta de entrada para acessar o caso de uso na API e o retorno dos dados, seja com qualquer uma das arquiteturas mencionadas. Além disso, a API foi desenvolvida com baixo acoplamento, utilizando os princípios SOLID e outros princípios da arquitetura limpa, que são princípios de design de código e que foram apresentados pela primeira vez pelo famoso cientista da computação Robert C. Martin no seu livro Arquitetura Limpa (MARTIN, 2019). Com a utilização do framework NestJS, com poucas linhas de código foi possível criar uma única API com a arquitetura REST, a linguagem GraphQL e com o framework gRPC. O código fonte da solução desenvolvida está disponível no apêndice C do trabalho.

Na Figura 21 temos a criação do servidor HTTP da API por meio de funções disponibilizadas pelo NestJS, a única configuração necessária para iniciar uma API com a arquitetura REST, sendo que as demais configurações são relacionadas aos princípios da arquitetura e dependem da implementação do desenvolvedor.

```
async function bootstrap() {
  const port = process.env.PORT || 3000
  const app = await NestFactory.create(AppModule, {
    logger: ['error', 'warn'],
    cors: { origin: '*', allowedHeaders: '*' },
  })
  await app.listen(port, () => {
    console.log(`Serviço executando na porta ${port}`)
  })
}

bootstrap()
```

Figura 21 - Criação do servidor HTTP da API no arquivo principal (main.ts)

Já para o funcionamento da linguagem GraphQL na API, foi necessário configurar o módulo GraphQL disponibilizado pelo NestJS na classe '**AppModule**' da aplicação, conforme Figura 22.

```
import { DatabaseModule } from './../core/infra/database/database.module.js';
import { MoviesModule } from './movies.module.js';
import { Module } from '@nestjs/common'
import { ConfigModule } from '@nestjs/config'
import { ScheduleModule } from '@nestjs/schedule'
import { GraphQLModule } from '@nestjs/graphql';
import { ApolloDriver, ApolloDriverConfig } from '@nestjs/apollo'
import { ApolloServerPluginLandingPageLocalDefault } from '@apollo/server/plugin/landingPage/default';
import { ActorsModule } from './actors.module.js';
import { CrewsModule } from './crews.module.js';
import { ReleasesModule } from './releases.module.js';

You, 4 weeks ago | 1 author (You)
@Module({
  imports: [
    ConfigModule.forRoot(),
    ScheduleModule.forRoot(),
    DatabaseModule,
    MoviesModule,
    ActorsModule,
    CrewsModule,
    ReleasesModule,
    GraphQLModule.forRoot<ApolloDriverConfig>({
      driver: ApolloDriver,
      autoSchemaFile: '../core/infra/graphql/schema.gql',
      playground: false,
      introspection: true,
      plugins: [ApolloServerPluginLandingPageLocalDefault() as any]
    }),
  ],
  controllers: [],
  providers: [],
})
export class AppModule {}
```

Figura 22 - Configuração do GraphQL na API destacado em vermelho (app.module.ts)

Como exemplo do funcionamento, na Figura 23 temos os arquivos com as classes responsáveis por executar o caso de uso '**ListMovies**' para cada tecnologia do estudo.

```
import { ListMovies } from '../../../../application/usecase/movies/list-movies.js';
import { Controller, Get, HttpStatus, ParseIntPipe, Query, Res, Version } from '@nestjs/common'
import { ApiOkResponse, ApiTags } from '@nestjs/swagger'
import { Response } from 'express';

You, last month | 1 author (You)
@ApiTags('Movies')
@Controller('movies')
export class RestListMoviesController {
  constructor(
    private readonly listMovies: ListMovies
  ) {}

  @ApiOkResponse({
    description: 'OK'
  })
  @Version('1')
  @Get()
  async handle(@Res() res: Response, @Query('limit', ParseIntPipe) limit: number): Promise<Response> {
    const result = await this.listMovies.execute(limit)
    return res.status(HttpStatus.OK).json(result);
  }
}
```

Figura 23 - Controlador para executar o caso de uso 'ListMovies' via REST

A API desenvolvida com o framework NestJS traz certas facilidades para trabalhar com todas as arquiteturas em estudo no presente trabalho, uma delas é a utilização de 'decorators' (decoradores), que segundo Osmani (2015) são basicamente expressões que retornam uma função e podem receber parâmetros (alvo, nome e descritor), a sintaxe é simples bastando aplicar um '@' como prefixo no nome do decorator, e podem ser aplicados em classes, métodos ou atributos. Na imagem anterior, por meio do decorator '@Controller()' e do decorator '@Get()' é possível criar uma rota com o método HTTP GET que é acessada por meio do recurso '/movies' no endereço da API.

```
import { ListMovies } from '../../../../application/usecase/movies/list-movies.js';
import { Args, Query, Resolver } from '@nestjs/graphql'
import { GraphQLError } from 'graphql'
import { MovieOutput } from '../../../../dto/movies.dto.js';
import { ParseIntPipe } from '@nestjs/common';

You, 4 weeks ago | 1 author (You)
@Resolver(() => Object)
export class GraphqlListMoviesResolver {
  constructor(private readonly listMovies: ListMovies) {}

  @Query(() => [MovieOutput], { name: 'list_movies', nullable: true })
  async handle(
    @Args('limit', ParseIntPipe) limit: number
  ) {
    try {
      return this.listMovies.execute(limit)
    } catch (error) {
      throw new GraphQLError('Erro desconhecido', {
        extensions: {
          code: 500
        }
      })
    }
  }
}
```

Figura 24 - Resolver para executar o caso de uso 'ListMovies' via GraphQL

Seguindo a mesma lógica, para a linguagem GraphQL, com NestJS basta aplicar o decorator '@Resolver()' a classe para que a mesma consiga fornecer instruções para transformar uma operação GraphQL (consulta, mutação ou assinatura) em dados. Os 'resolvers' retornam o mesmo modelo de dados que foram especificados no esquema GraphQL (NESTJS, 2024b). O esquema GraphQL com NestJS é gerado automaticamente, bastando apenas aplicar o decorator '@ObjectType()' em uma classe para definir o modelo de dados e o decorator '@Field()' para cada propriedade da classe, conforme exemplo da Figura 25:

```typescript
@ObjectType({ description: 'Modelo de dados de retorno um filme' })
export class MovieOutput {
  @ApiProperty({ description: 'Identificador', required: true })
  @Field((type) => Int, { description: 'Identificador', nullable: false })
  id: number

  @ApiProperty({ description: 'Nome', required: true })
  @Field((type) => String, { description: 'Nome', nullable: true })
  name: string

  @ApiProperty({ description: 'Ano de lançamento', required: true })
  @Field((type) => Int, { description: 'Ano de lançamento', nullable: true })
  date: number

  @ApiProperty({ description: 'Slogan', required: true })
  @Field((type) => String, { description: 'Slogan', nullable: true })
  tagline: string

  @ApiProperty({ description: 'Descrição', required: true })
  @Field((type) => String, { description: 'Descrição', nullable: true })
  description: string

  @ApiProperty({ description: 'Tempo de duração em minutos', required: true })
  @Field((type) => Int, { description: 'Tempo de duração em minutos', nullable: true })
  minute: number

  @ApiProperty({ description: 'Média de avaliação no Letterboxd', required: true })
  @Field((type) => Float, { description: 'Média de avaliação no Letterboxd', nullable: true })
  rating: number
}
```

Figura 25 - Classe para definir o modelo de dados GraphQL de um filme

```typescript
import { ListMovies } from '../../../../application/usecase/movies/list-movies.js';
import { Controller } from '@nestjs/common'
import { GrpcMethod } from '@nestjs/microservices'
import { Metadata, ServerUnaryCall } from '@grpc/grpc-js'

You, yesterday | 1 author (You)
@Controller()
export class GrpcListMoviesController {
  constructor(private readonly listMovies: ListMovies) {}

  @GrpcMethod('TccService', 'listMovies')
  async handle(
    data: { limit: number },
    metadata: Metadata,
    call: ServerUnaryCall<any, any>
  ): Promise<any> {
    const result = await this.listMovies.execute(Number(data.limit))
    return { movies: result }
  }
}
```

Figura 26 - Controlador para executar o caso de uso 'ListMovies' via gRPC

Para utilizar gRPC com NestJS, é necessário aplicar o decorator ‘@Controller()’ em uma classe para definir um controlador e o decorator ‘@GrpcMehod()’ para definir o método gRPC a ser executado, bastando passar como parâmetro o nome do serviço gRPC e o nome do método do serviço especificado, ambos definidos por meio de um arquivo proto configurado na aplicação. O arquivo proto da solução está disponível no apêndice B do trabalho.

5.4.1 TESTES REALIZADOS

A ferramenta k6 utilizada nos testes possui métricas gerais que são geradas sempre para qualquer protocolo que é testado e também métricas específicas para alguns protocolos, como é o caso do gRPC que foi utilizado no presente estudo.

As métricas gerais selecionadas para o presente estudo foram:

- **data_received**: quantidade de dados recebidos.
- **data_sent**: quantidade de dados enviados.
- **iteration_duration**: tempo para concluir uma iteração completa.
- **iterations**: número total de vezes que os usuários virtuais executaram o teste.

Para o protocolo HTTP as métricas selecionadas foram:

- **http_req_connecting**: tempo gasto para estabelecer conexão TCP com o servidor.
- **http_req_duration**: tempo total gasto na requisição.
- **http_req_receiving**: tempo gasto para receber dados de resposta do servidor.
- **http_req_sending**: tempo gasto no envio de dados para o servidor.
- **http_req_waiting**: tempo gasto aguardando a resposta do servidor
- **http_reqs**: número total de requisições HTTP geradas

Para o gRPC, somente a métrica ‘**grpc_req_duration**’ foi selecionada, que é o tempo total para receber resposta do servidor. Apesar do serviço gRPC utilizar o protocolo HTTP na versão 2, a ferramenta k6 não lista as métricas HTTP listadas anteriormente para um teste com gRPC, apenas as métricas gerais e as específicas para gRPC constam nos resultados.

Todos os testes foram realizados localmente em um computador conectado a uma internet com velocidade de 200 Mbps com a seguinte configuração:

- Processador Intel(R) Core(TM) i5-10400F CPU @ 2.90GHz 2.90 GHz
- Memória RAM de 16GB
- Sistema Operacional Windows 11 Home de 64 bits
- Placa de vídeo NVIDIA GeForce RTX 2060 SUPER
- SSD KINGSTON de 240GB

Todos os testes de criação e listagem foram realizados com uma configuração para simular 10 usuários virtuais e por um tempo de utilização de 30 segundos, já os testes de edição e remoção foram realizados com uma configuração para simular 1 usuário virtual por um tempo de utilização de 30 segundos, pois devido a necessidade de enviar identificadores como parâmetro, não foi possível configurar para que usuários virtuais diferentes enviassem identificadores diferentes. Além disso, todos os testes de listagem foram realizados com quatro quantidades de itens fixas, sendo respectivamente 1.000, 10.000, 150.000 e 400.000 itens, exceto com gRPC que no cenário com 400.000 itens não foi possível executar devido ao limite de dados recebidos em bytes pela ferramenta de teste. Os casos de uso selecionados para a realização dos testes foram ‘**ListMovies**’ (listagem de filmes), ‘**AddMovie**’ (cadastro de um filme), ‘**EditMovie**’ (edição de dados de um filme) e ‘**DeleteMovie**’ (exclusão de um filme).

Em todos os cenários de testes que foram executados, os testes foram repetidos múltiplas vezes para gerar uma maior confiabilidade nos resultados do

estudo. Para o presente capítulo os resultados foram apresentados com apenas três métricas, sendo o número total de iterações, o tempo médio de cada iteração e o tempo médio aguardando a resposta do servidor, pois essas métricas são melhores para um comparativo geral de desempenho. Os resultados completos com todas as métricas destacadas na presente seção e que foram retornados pela ferramenta de testes estão disponíveis no apêndice A do trabalho.

1) Teste do caso de uso de listagem de filmes

```
import { Injectable, Inject } from '@nestjs/common'
import { MoviesRepository } from '../../repository/movies.repository.js'
import { Movie } from '../../../../core/domain/entity/movie.js'

You, last month | 1 author (You)
@Injectable()
export class ListMovies {
  constructor(
    @Inject('MoviesRepository')
    private moviesRepository: MoviesRepository
  ) {}

  async execute(limit: number): Promise<Movie[]> {
    return this.moviesRepository.find(limit)
  }
}
```

Figura 27 - Caso de uso 'ListMovies'

O caso de uso '**ListMovies**' realiza uma busca de filmes no banco de dados de acordo com a quantidade passada pelo parâmetro numérico '**limit**', exemplificando, caso seja informado o valor 1000 para o respectivo parâmetro, o retorno contará com dados de 1000 filmes.

Tabela 4 - Comparativo de desempenho do caso de uso 'ListMovies' com 1.000 itens

Quantidade de dados	Arquitetura	Total de iterações	Tempo médio de cada iteração	Tempo médio aguardando resposta do servidor
1.000	REST	2651	113.36 ms	112.7 ms
1.000	GraphQL	1375	218.93 ms	218.25 ms
1.000	gRPC	2696	111.48 ms	77.08 ms

Tabela 5 - Comparativo de desempenho do caso de uso 'ListMovies' com 10.000 itens

Quantidade de dados	Arquitetura	Total de iterações	Tempo médio de cada iteração	Tempo médio aguardando resposta do servidor
10.000	REST	238	1.28 s	1.28 s
10.000	GraphQL	139	2.24 s	2.23 s
10.000	gRPC	234	1.28 s	1.12 s

Tabela 6 - Comparativo de desempenho do caso de uso 'ListMovies' com 150.000 itens

Quantidade de dados	Arquitetura	Total de iterações	Tempo médio de cada iteração	Tempo médio aguardando resposta do servidor
150.000	REST	21	17.02 s	16.95 s
150.000	GraphQL	17	26.27 s	26.19 s
150.000	gRPC	20	24.24 s	22.05 s

Tabela 7 - Comparativo de desempenho do caso de uso 'ListMovies' com 400.000 itens

Quantidade de dados	Arquitetura	Total de iterações	Tempo médio de cada iteração	Tempo médio aguardando resposta do servidor
400.000	REST	8	44.2 s	43.87 s
400.000	GraphQL	4	45.08 s	44.75 s
400.000	gRPC	--	--	--

Com base nos quatro cenários de teste para o caso de uso de listagem de filmes, como resultado REST obteve um desempenho superior com 10.000, 150.000 e 400.000 itens retornados. Já o gRPC obteve o melhor desempenho no cenário com 1.000 itens retornados.

2) Teste do caso de uso de cadastro de um filme

```
import { Injectable, Inject } from '@nestjs/common'
import { MoviesRepository } from '../../repository/movies.repository.js'
import { MovieInput } from '../../../../core/dto/movies.dto.js'
import { Movie } from '../../../../core/domain/entity/movie.js'

You, last month | 1 author (You)
@Injectable()
export class AddMovie {
  constructor(
    @Inject('MoviesRepository')
    private moviesRepository: MoviesRepository
  ) {}

  async execute(movie_input: MovieInput): Promise<Movie> {
    const movie = Movie.create(movie_input)
    const result = await this.moviesRepository.create(movie)
    return result
  }
}
```

Figura 28 - Caso de uso 'AddMovie'

O caso de uso ‘**AddMovie**’ é responsável pelo cadastro de um filme e recebe como parâmetro os dados necessários para cadastrar um filme, cria o objeto de um filme e faz o registro no banco de dados.

Tabela 8 - Comparativo de desempenho do caso de uso 'AddMovie'

Arquitetura	Total de iterações	Tempo médio de cada iteração	Tempo médio aguardando resposta do servidor
REST	17.922	16.73 ms	16.54 ms
GraphQL	30.359	9.86 ms	9.65 ms
gRPC	13.162	22.78 ms	13.94 ms

No caso de uso de cadastro de filmes, GraphQL obteve um desempenho bastante superior em relação a REST e gRPC, com praticamente o dobro de inserções.

3) Teste do caso de uso de edição de dados de um filme

```
import { Injectable, Inject } from '@nestjs/common'
import { MoviesRepository } from '../../repository/movies.repository.js'
import { MovieInput } from '../../../dto/movies.dto.js'
import { Movie } from '../../../domain/entity/movie.js'

You, last month | 1 author (You)
@Injectable()
export class EditMovie {
  constructor(
    @Inject('MoviesRepository')
    private moviesRepository: MoviesRepository
  ) {}

  async execute(movie_id: number, movie_input: MovieInput): Promise<Movie> {
    const movie = await this.moviesRepository.findById(movie_id)
    movie.data = movie_input
    return this.moviesRepository.edit(movie)
  }
}
```

Figura 29 - Caso de uso 'EditMovie'

O caso de uso '**EditMovie**' é responsável por editar dados de um filme e recebe como parâmetro o identificador do filme e também os dados do filme que serão atualizados. É realizado uma busca do filme existente no banco de dados pelo identificador recebido como parâmetro e na sequência os dados do filme são atualizados no banco de dados.

Tabela 9 - Comparativo de desempenho do caso de uso 'EditMovie'

Arquitetura	Total de iterações	Tempo médio de cada iteração	Tempo médio aguardando resposta do servidor
REST	3.894	7.66 ms	7.12 ms
GraphQL	3.611	8.26 ms	7.72 ms
gRPC	3.238	9.26 ms	8.30 ms

Os resultados obtidos no caso de uso de edição de um filme demonstraram um maior desempenho com a arquitetura REST, porém tanto GraphQL como gRPC apresentaram um desempenho próximo a REST.

4) Teste do caso de uso de exclusão de um filme

```
import { Injectable, Inject } from '@nestjs/common'
import { MoviesRepository } from '../../repository/movies.repository.js'

You, last month | 1 author (You)
@Injectable()
export class DeleteMovie {
  constructor(
    @Inject('MoviesRepository')
    private moviesRepository: MoviesRepository
  ) {}

  async execute(movie_id: number): Promise<String> {
    return this.moviesRepository.delete(movie_id)
  }
}
```

Figura 30 - Caso de uso 'DeleteMovie'

O caso de uso '**DeleteMovie**' é responsável por excluir um filme e recebe como parâmetro o identificador de um filme. Com base no identificador recebido, o filme é removido do banco de dados.

Tabela 10 - Comparativo de desempenho do caso de uso 'DeleteMovie'

Arquitetura	Total de iterações	Tempo médio de cada iteração	Tempo médio aguardando resposta do servidor
REST	22.830	1.31 ms	832.99 µs
GraphQL	21.369	1.35 ms	790.62 µs
gRPC	15.473	1.93 ms	1.3 ms

No caso de uso de exclusão de filmes, REST obteve um melhor desempenho e GraphQL obteve um desempenho bastante próximo. A arquitetura gRPC ficou um pouco distante em relação aos demais nesse cenário de teste.

5.4.2 COMPARATIVO

Todos os testes foram realizados com sucesso, exceto a listagem com 400.000 itens que não foi possível realizar com gRPC devido a uma limitação de recebimento de dados pela ferramenta de testes, mesmo consultando a documentação não foi possível solucionar e por isso para o cenário de teste de listagem com a maior quantidade de dados com gRPC foi possível listar somente 150.000 itens, o que excluiu a arquitetura da comparação com REST e GraphQL nesse cenário.

Em relação ao cadastro de itens, GraphQL obteve o melhor resultado com uma média de 9.86 milissegundos por iteração e realizando 30.359 cadastros em um período de 30 segundos, REST ficou em segundo lugar com 17.922 cadastros realizados e uma média de 16.73 milissegundos por iteração e gRPC em terceiro lugar com 13.162 cadastros e uma média de 22.78 milissegundos por iteração.

Em relação a edição de itens, REST obteve o melhor desempenho com 3.894 iterações e uma média de 7.66 milissegundos, GraphQL na sequência obteve 3.611 iterações e uma média de 8.26 milissegundos e gRPC obteve 3.238 iterações com uma média de 9.26 milissegundos.

Em relação a exclusão de itens, REST também obteve o melhor desempenho com 22.830 iterações e uma média de 1.31 milissegundos, GraphQL ficou muito próximo na sequência com 21.369 iterações e uma média de 1.35 milissegundos e gRPC obteve 15.473 iterações com uma média de 1.93 milissegundos.

Por fim, em relação a listagem de itens, no cenário com 1.000 itens o gRPC obteve melhor resultado com 2.696 iterações e uma média de 111.48 milissegundos por iteração, REST obteve o segundo melhor resultado com 2.651

iterações e uma média de 113.35 milissegundos por iteração e GraphQL obteve o terceiro lugar com 1.375 iterações e uma média de 218.93 milissegundos por iteração. No cenário com 10.000 itens, REST obteve o melhor desempenho com 238 iterações e uma média por iteração de 1.28 segundos, gRPC ficou muito próximo com 234 iterações e também uma média por iteração de 1.28 segundos e GraphQL novamente obteve o terceiro lugar com 139 iterações e uma média por iteração de 2.24 segundos. No terceiro cenário com 150.000 itens, REST obteve o melhor desempenho com 21 iterações e uma média de 17.02 segundos por iteração, gRPC ficou muito próximo e obteve um resultado de 20 iterações com uma média por iteração de 24.24 segundos e GraphQL ficou em terceiro lugar, mas com um resultado bem próximo com 17 iterações e uma média de 26.27 segundos por iteração. Na disputa do último cenário entre REST e GraphQL com 400.000 itens, REST obteve o melhor resultado com 8 iterações e uma média por iteração de 44.2 segundos contra 4 iterações e uma média de 45.08 segundos da linguagem GraphQL.

Com base nos resultados, temos que REST obteve o melhor desempenho no geral pois realizou mais exclusões, edições e também ganhou em 3 dos 4 cenários de teste de listagem. Na sequência temos a linguagem GraphQL que obteve o melhor desempenho em relação ao cadastro, com uma diferença grande de 12.437 itens a mais em relação a REST e 17.197 itens a mais em relação ao gRPC, além de ser superior a gRPC nas edições e exclusões, tendo o pior desempenho apenas nas listagens. Apesar de gRPC obter o terceiro melhor desempenho no cenário geral, vale destacar que em relação às listagens, gRPC obteve o melhor desempenho com 1.000 itens e praticamente empatou com REST nos cenários de 10.000 e 150.000 itens com a mesma média de tempo por iteração e 4 iterações de diferença no total no cenário de 10.000 itens e apenas uma iteração de diferença no cenário com 150.000 itens. Conforme já destacado na presente seção, devido a limitação de dados não foi possível executar o cenário de teste com 400.000 itens com gRPC.

6 CONCLUSÃO

Por meio deste trabalho, foi possível entender o funcionamento geral de cada arquitetura em estudo, com suas respectivas características, modo de funcionamento, vantagens, desvantagens e informações gerais, como a documentação e sua utilização pela comunidade de desenvolvimento. Além disso, foi realizado um comparativo entre as três arquiteturas analisados em todos os aspectos citados anteriormente e também entre o desempenho de cada uma por meio de testes de carga realizados em cima da API desenvolvida no estudo.

A ideia principal do estudo foi a realização do comparativo sem eleger uma arquitetura de desenvolvimento de API como sendo a melhor em relação as outras, mas sim buscando destacar a diferença de cada uma e para quais cenários de uso cada arquitetura encaixa melhor.

Os resultados apresentados pela solução implementada, demonstraram que dependendo do caso de uso executado pela API e/ou cenário de teste executado, cada arquitetura de desenvolvimento desempenhou ou não melhor que o outro, sem demonstrar um padrão de resultados onde somente uma arquitetura é superior às demais. Em testes relacionados a inserção de dados, GraphQL foi superior em relação às outras. Já a arquitetura gRPC ficou muito próxima a REST nos cenários de listagem, tendo inclusive um melhor desempenho no teste de listagem com menor quantidade de dados. A arquitetura REST obteve o melhor desempenho nos demais testes.

Como limitações do estudo, temos que a execução da solução e dos testes foram realizados exclusivamente em um ambiente local com uma rede estável e de baixa latência, o que não ocorre geralmente nos cenários tradicionais pois os projetos precisam estar hospedados em servidores que passam por instabilidades, dependem de navegadores de internet (browsers), aplicativos móveis, entre outros fatores que podem afetar o funcionamento e desempenho da aplicação. Além disso, a ferramenta de testes utilizada poderia ter sido mais explorada com mais cenários de testes e a API desenvolvida por

mais que tenha sido desenvolvida com padrões de design de código, utilizou frameworks e bibliotecas que podem ou não ter influenciado nos resultados.

Por fim, apesar de REST dominar o cenário de desenvolvimento back-end até os dias atuais e ter obtido o melhor desempenho nos resultados do presente trabalho, este estudo traz como principal contribuição para o mercado de desenvolvimento a demonstração que há uma crescente busca de novas tecnologias, como GraphQL e gRPC, com o intuito de melhorar ou trazer novas opções que podem e já estão sendo aplicadas em diversos projetos.

6.1 TRABALHOS FUTUROS

Com base nas limitações apresentadas na seção anterior e nos estudos realizados no presente trabalho, os possíveis trabalhos futuros podem ser desenvolvidos:

- Hospedagem da API desenvolvida em um servidor e disponibilizada em algum domínio.
- Exploração da ferramenta de testes Grafana k6 com criação de mais cenários de teste para realização dos testes de carga da API desenvolvida.
- Adicionar novos critérios de comparação nos testes aplicados.
- Criação de uma nova API sem utilização de frameworks como os utilizados na solução do presente estudo e também com outros frameworks para comparação entre os testes de desempenho entre essas APIs.
- Estudo de outras arquiteturas, padrões ou tecnologias para comparação com as do presente estudo.
- Implementação de suporte à requisições de fluxo contínuo (streaming) na API da solução para fins de teste e comparação.

REFERÊNCIAS

AL-KHAMISI, Zakariya. Top 6 API Architecture Styles for Modern Software Development. 2023. Disponível em: https://zikazaki.medium.com/top-6-api-architecture-styles-for-modern-software-development-d35752c7f9aa. Acesso em: 10 dez. 2023.

AWS. **Qual é a diferença entre front-end e back-end no desenvolvimento de aplicações?** 2023a. Disponível em: https://aws.amazon.com/pt/compare/the-difference-between-frontend-and-backend/. Acesso em: 15 nov. 2023.

AWS. O que é uma API (interface de programação de aplicações)? 2023b. Disponível em: https://aws.amazon.com/pt/what-is/api/#:~:text=API%20significa%20Application%20Programming%20Interface,de%20servi%C3%A7o%20entre%20duas%20aplica%C3%A7%C3%B5es. Acesso em: 19 nov. 2023.

AWS. Https://aws.amazon.com/caching/: caching helps applications perform dramatically faster and cost significantly less at scale. Caching helps applications perform dramatically faster and cost significantly less at scale. 2023c. Disponível em: https://aws.amazon.com/caching/. Acesso em: 19 nov. 2023

BRONZE, Bruno Oliveira; FARINA, Renata Mirella; FLORIAN, Fabiana. GRPC (GOOGLE REMOTE PROCEDURE CALL) ESTRUTURA UNIVERSAL DE CÓDIGO ABERTO. Recima21 - Revista Científica Multidisciplinar - Issn 2675-6218, [S.L.], v. 4, n. 1, p. 1-6, 10 nov. 2023. RECIMA21 - Revista Cientifica Multidisciplinar. http://dx.doi.org/10.47820/recima21.v4i1.4492.

BRASSCOM. Demanda de Talentos em TIC e Estratégia Σ TCEM. 2021. Disponível em: https://brasscom.org.br/pdfs/demanda-de-talentos-em-tic-e-estrategia-tcem/. Acesso em: 25 nov. 2023.

BUňATA, Tomáš. **Srovnání REST, GraphQL a gRPC API v Node.js**. 2022. 95 f. Tese (Doutorado) - Curso de Software Engineering, Department Of Software Engineering, Faculty Of Information Technology Ctu, Praga, 2022.

DELLANO, Klaus. As 6 Arquiteturas de API mais procuradas. 2023. Disponível em: https://medium.com/havanlabs/as-6-arquiteturas-de-api-mais-procuradas-25128fdbcd72. Acesso em: 10 dez. 2023.

FIELDING, Roy Thomas. **Architectural Styles and the Design of Network-based Software Architectures**. 2000. 180 f. Dissertação (Doutorado) - Curso de Information And Computer Science, University Of California, Irvine, 2000.

GARANIN, Simon. **Letterboxd**: more than 850,000 films with descriptions, posters, actors, crew and releases. More than 850,000 films with descriptions, posters, actors, crew and releases. 2024. Disponível em: https://www.kaggle.com/datasets/gsimonx37/letterboxd. Acesso em: 10 abr. 2024.

GRPC. **GRPC**: a high performance, open source universal rpc framework get started!. 2024a. Disponível em: https://grpc.io/. Acesso em: 05 maio 2024.

GRPC. **About gRPC**: who is using grpc and why. 2024b. Disponível em: https://grpc.io/about/. Acesso em: 05 maio 2024.

GRPC. **Introduction to gRPC**: an introduction to grpc and protocol buffers. 2024c. Disponível em: https://grpc.io/docs/what-is-grpc/introduction/. Acesso em: 05 maio 2024.

GRPC. **Core concepts, architecture and lifecycle**: an introduction to key grpc concepts, with an overview of grpc architecture and rpc life cycle. 2024d. Disponível em: https://grpc.io/docs/what-is-grpc/core-concepts/. Acesso em: 05 maio 2024.

GRPC. **Authentication**: an overview of grpc authentication, including built-in auth mechanisms, and how to plug in your own authentication systems. 2024e. Disponível em: https://grpc.io/docs/guides/auth/. Acesso em: 20 maio 2024

GRPC. **Community**. 2024e. Disponível em: https://grpc.io/community/. Acesso em: 10 maio 2024.

GRPC. **Documentation**. 2024f. Disponível em: https://grpc.io/docs/. Acesso em: 10 maio 2024.

GRAPHQL. **GraphQL**. 2024a. Disponível em: https://spec.graphql.org/draft/#sec-Overview. Acesso em: 10 abr. 2024.

GRAPHQL. What is the GraphQL Foundation? 2024b. Disponível em: https://graphql.org/community/foundation/. Acesso em: 11 abr. 2024.

GRAPHQL. Introduction to GraphQL. 2024c. Disponível em: https://graphql.org/learn/. Acesso em: 25 abr. 2024.

HOSTGATOR. API: Entenda a Interface de Programação de Aplicação. 2020. Disponível em: https://www.hostgator.com.br/blog/api-entenda-a-interface-de-programacao-de-aplicacao/. Acesso em: 10 dez. 2023.

IBM. **Remote Procedure Call**. 2024. Disponível em: https://www.ibm.com/docs/en/aix/7.3?topic=concepts-remote-procedure-call. Acesso em: 05 maio 2024.

INSOMNIA. Design, Debug, Test & Mock APIs Locally, on Git or Cloud.: build better apis collaboratively for the most popular protocols with a dev-friendly ui, built-in automation, and an extensible plugin ecosystem. 2024a. Disponível em: https://insomnia.rest/. Acesso em: 26 abr. 2024.

INSOMNIA. Welcome to Insomnia Docs: all the information needed to deliver high-quality apis through standards and collaboration with the insomnia api design platform. 2024b. Disponível em: https://docs.insomnia.rest/. Acesso em: 26 abr. 2024.

K6. **The best developer experience for load testing**. 2024. Disponível em: https://k6.io/. Acesso em: 11 maio 2024.

LANGUAGE Guide (proto 3): Covers how to use the version 3 of Protocol Buffers in your project. 2024. Disponível em: https://protobuf.dev/programming-guides/proto3/. Acesso em: 05 maio 2024.

MARTIN, Robert C.. **Arquitetura Limpa**: o guia do artesão para estrutura e design de software. Rio de Janeiro: Alta Books, 2019. 432 p. Traduzido por: Samantha Batista.

MONGOOSE. Mongoose: elegant mongodb object modeling for node.js. 2024. Disponível em: https://mongoosejs.com/. Acesso em: 25 abr. 2024.

MONGODB. What is MongoDB? 2024. Disponível em: https://www.mongodb.com/docs/manual/. Acesso em: 25 abr. 2024.

MDN WEB DOCS. **An overview of HTTP**. 2024a. Disponível em: https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview. Acesso em: 10 maio 2024.

MDN WEB DOCS. URL. 2024b. Disponível em: https://developer.mozilla.org/pt-BR/docs/Glossary/URL. Acesso em: 25 maio 2024.

MURPHY, Lauren *et al*. **Preliminary Analysis of REST API Style Guidelines**. 2017. Computer Science, Vancouver, 2017.

NESTJS. Hello, nest!: a progressive node.js framework for building efficient, reliable and scalable server-side applications. 2024a. Disponível em: https://nestjs.com/. Acesso em: 25 abr. 2024.

NESTJS. Resolvers. 2024b. Disponível em: https://docs.nestjs.com/graphql/resolvers. Acesso em: 10 maio 2024.

NODE.JS. Run JavaScript Everywhere: node.js® is a free, open-source, cross-platform javascript runtime environment that lets developers create servers, web apps, command line tools and scripts. 2024. Disponível em: https://nodejs.org/en. Acesso em: 25 abr. 2024.

ORACLE. O que é NoSQL? 2024. Disponível em: https://www.oracle.com/br/database/nosql/what-is-nosql/. Acesso em: 25 abr. 2024.

OLAWANLE, Joel. What is a Framework? Software Frameworks Definition. 2022. Disponível em: https://www.freecodecamp.org/news/what-is-a-framework-software-frameworks-definition/. Acesso em: 26 nov. 2023.

OSMANI, Addy. Exploring EcmaScript Decorators. 2015. Disponível em: https://medium.com/google-developers/exploring-es7-decorators-76ecb65fb841. Acesso em: 20 maio 2024.

POSTMAN. 2023 State of the API Report. 2023. Disponível em: https://www.postman.com/state-of-api/. Acesso em: 09 abr. 2024.

QUIÑA-MERA, Antonio; FERNANDEZ, Pablo; GARCÍA, José María; RUIZ-CORTÉS, Antonio. GraphQL: a systematic mapping study. Acm Computing Surveys, [S.L.], v. 55, n. 10, p. 1-35, 2 fev. 2023. Association for Computing Machinery (ACM). http://dx.doi.org/10.1145/3561818.

RED HAT. O que é API? 2023. Disponível em: https://www.redhat.com/pt-br/topics/api/what-are-application-programming-interfaces. Acesso em: 10 dez. 2023.

STACK OVERFLOW. **Stack Overflow**. Disponível em: https://stackoverflow.com/. Acesso em: 10 abr. 2024.

STATCOUNTER. Search Engine Market Share Worldwide - April 2024. 2024. Disponível em: https://gs.statcounter.com/search-engine-market-share. Acesso em: 25 maio 2024.

TYPESCRIPT. TypeScript is JavaScript with syntax for types.: typescript is a strongly typed programming language that builds on javascript, giving you better tooling at any scale. 2024. Disponível em: https://www.typescriptlang.org/. Acesso em: 20 abr. 2024.

Fonte: Wallarm. Disponível em: https://www.wallarm.com/what/what-is-http-2-and-how-is-it-different-from-http-1. Acesso em: 15 maio 2024.

APÊNDICE A – RESULTADOS DOS TESTES COM FERRAMENTA GRAFANA K6

A.1 - Caso de uso de listagem de filmes

REST

Teste com 1000 filmes

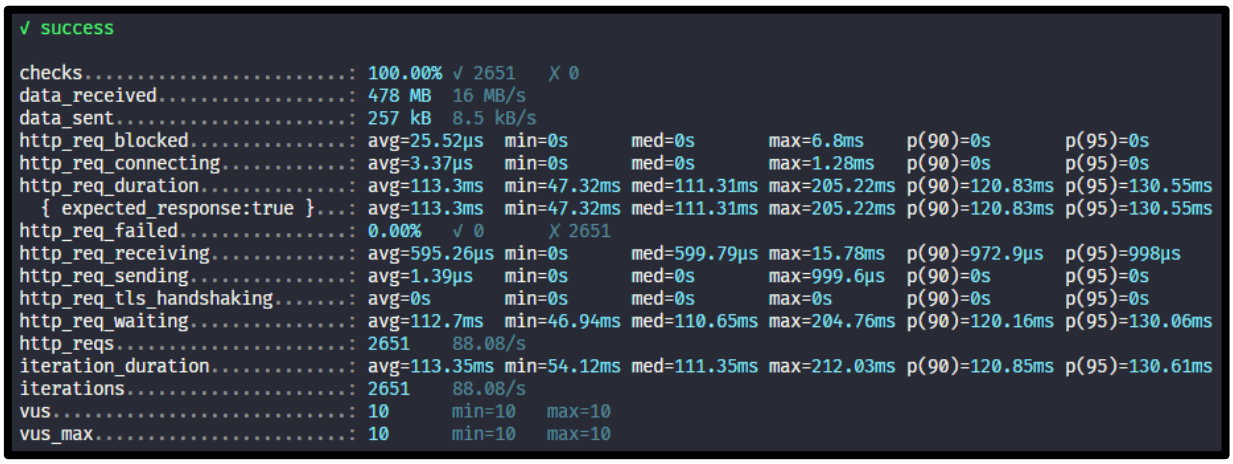


Figura 31 - Resultado completo do teste de listagem de 1.000 filmes utilizando REST

O resultado desse teste apresentou uma quantidade de 257 kB de dados enviados e 478 MB de dados recebidos, com um total de 2651 iterações/requisições HTTP, tempo médio de cada iteração de 113.35 milissegundos, tempo médio de conexão HTTP de 3.37 microssegundos, tempo médio de duração total da requisição HTTP de 113.3 milissegundos, tempo médio gasto para receber dados de resposta do servidor de 595.26 microssegundos, tempo médio gasto para enviar dados para o servidor de 1.39 microssegundos e tempo médio gasto aguardando a resposta do servidor de 112.7 milissegundos.

Teste com 10.000 filmes

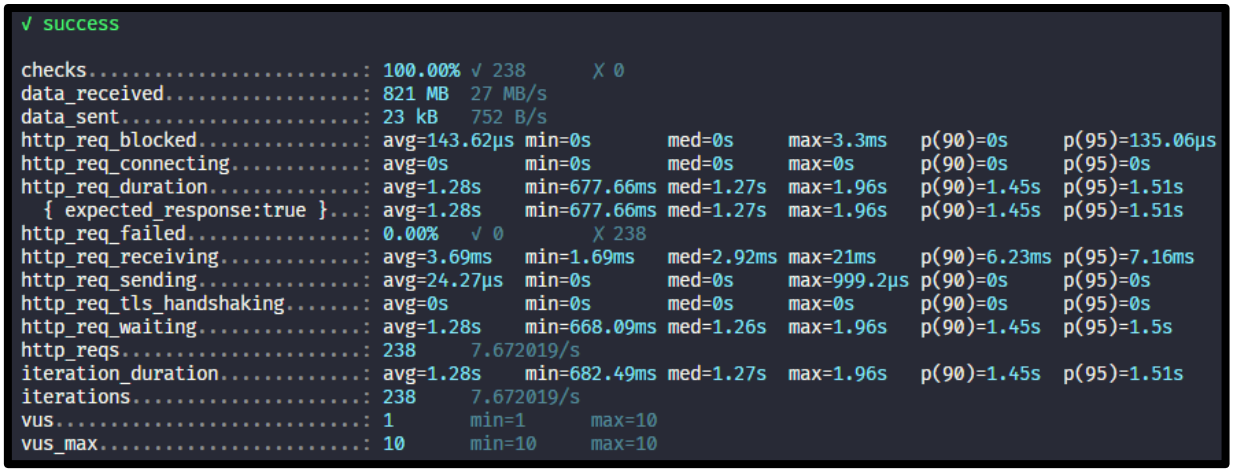


Figura 32 - Resultado completo do teste de listagem de 10.000 filmes utilizando REST

O resultado desse teste apresentou uma quantidade de 23 kB de dados enviados e 821 MB de dados recebidos, com um total de 238 iterações/requisições HTTP, tempo médio de cada iteração de 1.28 segundos, tempo médio de conexão HTTP de 0 segundos, tempo médio de duração total da requisição HTTP de 1.28 segundos, tempo médio gasto para receber dados de resposta do servidor de 3.69 milissegundos, tempo médio gasto para enviar dados para o servidor de 24.27 microssegundos e tempo médio gasto aguardando a resposta do servidor de 1.28 segundos.

Teste com 150.000 filmes

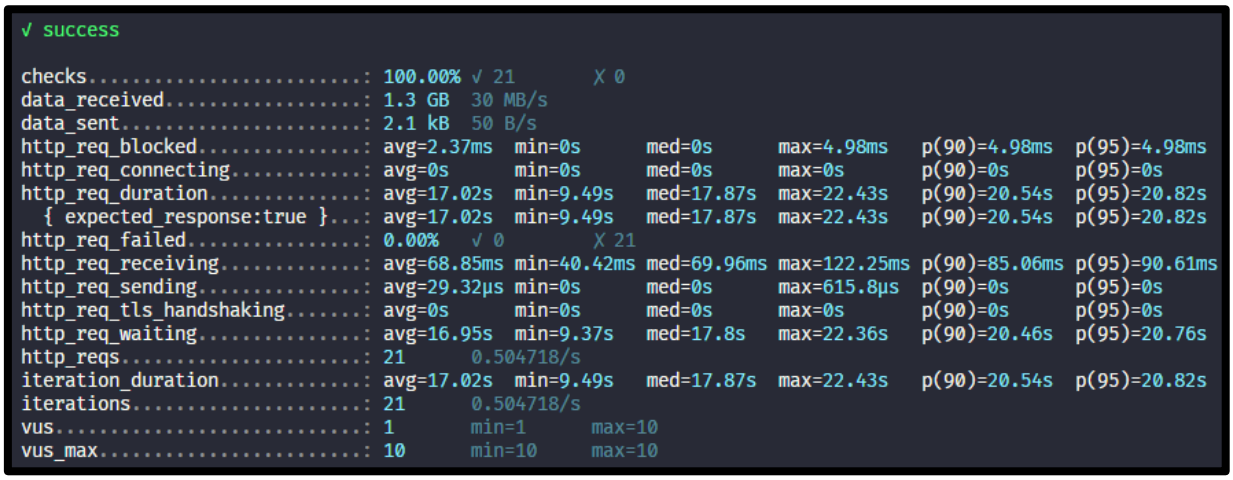


Figura 33 - Resultado completo do teste de listagem de 150.000 filmes utilizando REST

O resultado desse teste apresentou uma quantidade de 2.1 kB de dados enviados e 1.3 GB de dados recebidos, com um total de 21 iterações/requisições HTTP, tempo médio de cada iteração de 17.02 segundos, tempo médio de conexão HTTP de 0 segundos, tempo médio de duração total da requisição HTTP de 17.02 segundos, tempo médio gasto para receber dados de resposta do servidor de 68.85 milissegundos, tempo médio gasto para enviar dados para o servidor de 29.32 microssegundos e tempo médio gasto aguardando a resposta do servidor de 16.95 segundos.

Teste com 400.000 filmes

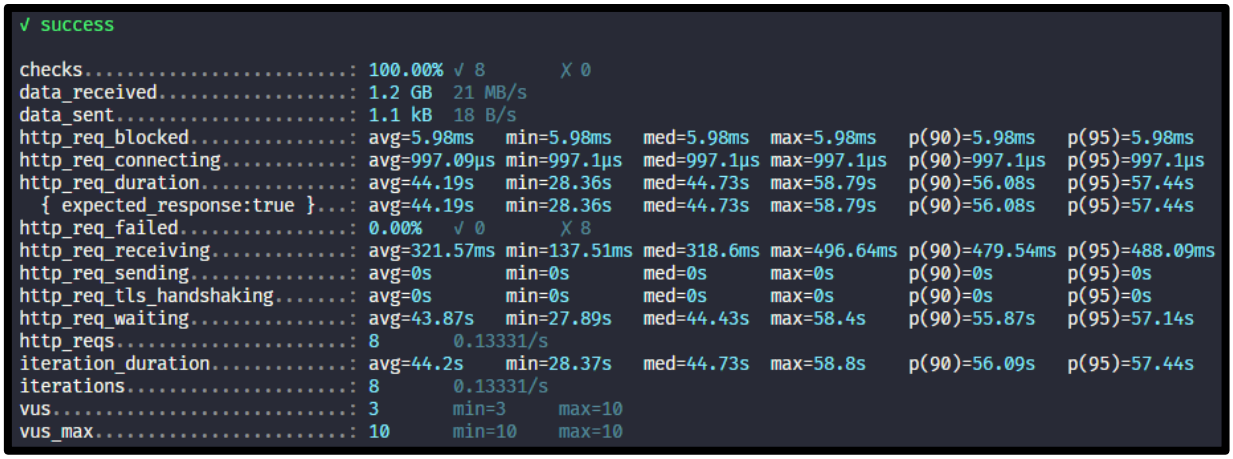


Figura 34 - Resultado completo do teste de listagem de 400.000 filmes utilizando REST

O resultado desse teste apresentou uma quantidade de 1.1 kB de dados enviados e 1.2 GB de dados recebidos, com um total de 8 iterações/requisições HTTP, tempo médio de cada iteração de 44.2 segundos, tempo médio de conexão HTTP de 997.09 microssegundos, tempo médio de duração total da requisição HTTP de 44.19 segundos, tempo médio gasto para receber dados de resposta do servidor de 321.57 milissegundos, tempo médio gasto para enviar dados para o servidor de 0 segundos e tempo médio gasto aguardando a resposta do servidor de 43.87 segundos.

GraphQL

Teste com 1000 filmes

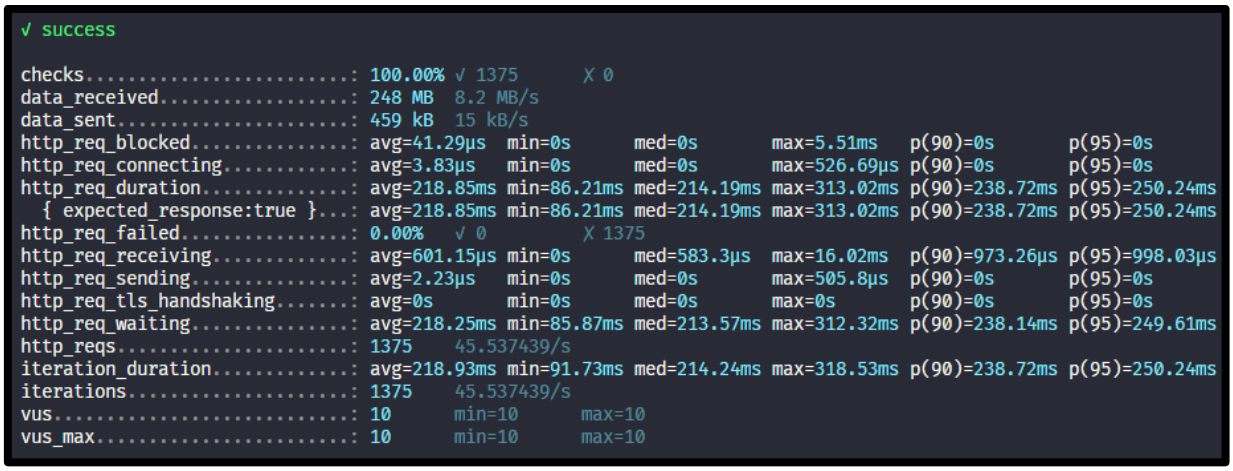


Figura 35 - Resultado completo do teste de listagem de 1.000 filmes utilizando GraphQL

O resultado desse teste apresentou uma quantidade de 459 kB de dados enviados e 248 MB de dados recebidos, com um total de 1375 iterações/requisições HTTP, tempo médio de cada iteração de 218.93 milissegundos, tempo médio de conexão HTTP de 3.83 microssegundos, tempo médio de duração total da requisição HTTP de 218.85 milissegundos, tempo médio gasto para receber dados de resposta do servidor de 601.15 microssegundos, tempo médio gasto para enviar dados para o servidor de 2.23 microssegundos e tempo médio gasto aguardando a resposta do servidor de 218.25 milissegundos.

Teste com 10.000 filmes

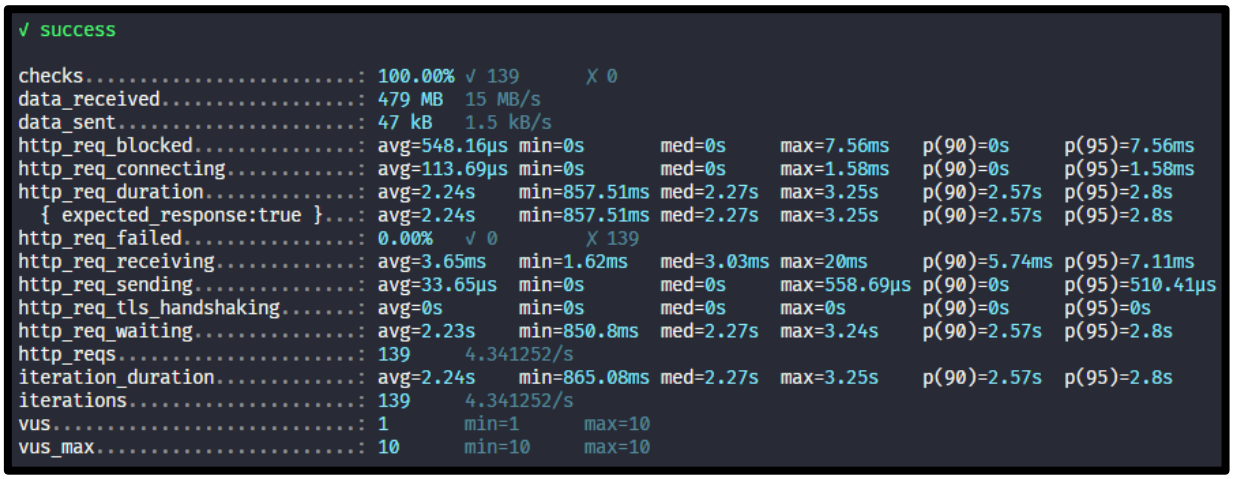


Figura 36 - Resultado completo do teste de listagem de 10.000 filmes utilizando GraphQL

O resultado desse teste apresentou uma quantidade de 47 kB de dados enviados e 479 MB de dados recebidos, com um total de 139 iterações/requisições HTTP, tempo médio de cada iteração de 2.24 segundos, tempo médio de conexão HTTP de 113.69 microssegundos, tempo médio de duração total da requisição HTTP de 2.24 segundos, tempo médio gasto para receber dados de resposta do servidor de 3.65 milissegundos, tempo médio gasto para enviar dados para o servidor de 33.65 microssegundos e tempo médio gasto aguardando a resposta do servidor de 2.23 segundos.

Teste com 150.000 filmes

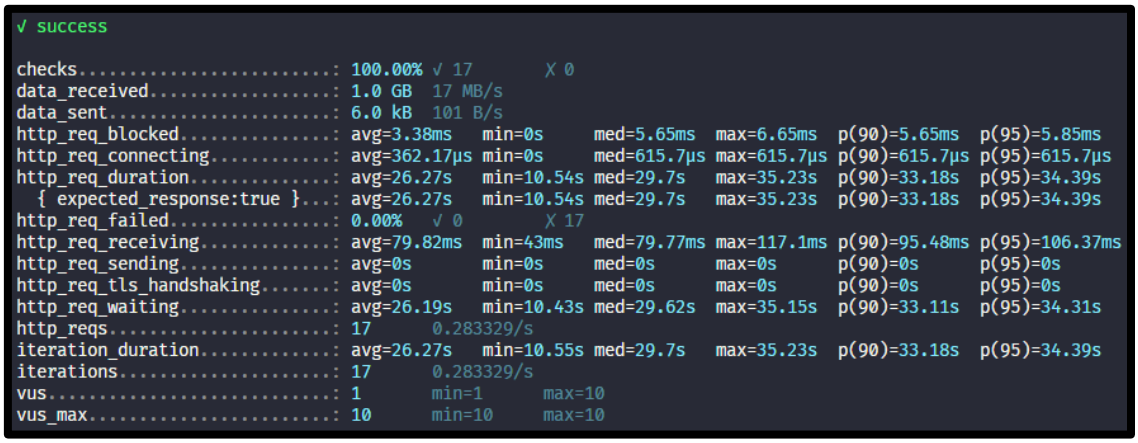


Figura 37 - Resultado completo do teste de listagem de 150.000 filmes utilizando GraphQL

O resultado desse teste apresentou uma quantidade de 6 kB de dados enviados e 1 GB de dados recebidos, com um total de 17 iterações/requisições HTTP, tempo médio de cada iteração de 26.27 segundos, tempo médio de conexão HTTP de 362.17 microssegundos, tempo médio de duração total da requisição HTTP de 26.27 segundos, tempo médio gasto para receber dados de resposta do servidor de 79.82 milissegundos, tempo médio gasto para enviar dados para o servidor de 0 segundos e tempo médio gasto aguardando a resposta do servidor de 26.19 segundos.

Teste com 400.000 filmes

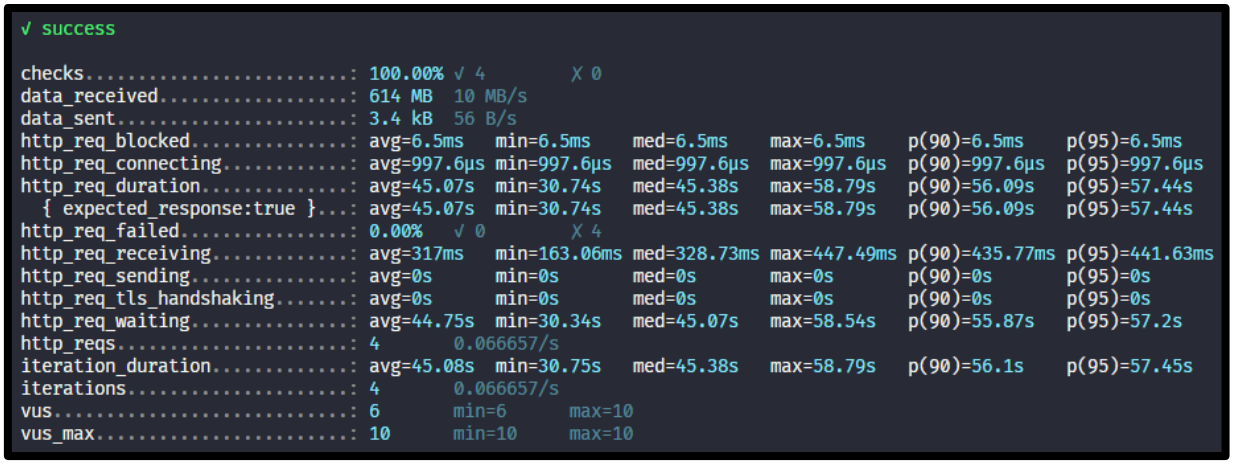


Figura 38 - Resultado completo do teste de listagem de 400.000 filmes utilizando GraphQL

O resultado desse teste apresentou uma quantidade de 3.4 kB de dados enviados e 614 MB de dados recebidos, com um total de 4 iterações/requisições HTTP, tempo médio de cada iteração de 45.08 segundos, tempo médio de conexão HTTP de 997.6 microssegundos, tempo médio de duração total da requisição HTTP de 45.07 segundos, tempo médio gasto para receber dados de resposta do servidor de 317 milissegundos, tempo médio gasto para enviar dados para o servidor de 0 segundos e tempo médio gasto aguardando a resposta do servidor de 44.75 segundos.

gRPC

Teste com 1000 filmes

```
✓ status is OK

checks.............: 100.00% ✓ 2696      ✗ 0
data_received.....: 271 MB  9.0 MB/s
data_sent.........: 857 kB  29 kB/s
grpc_req_duration....: avg=77.08ms  min=28.78ms med=76.5ms   max=177.44ms p(90)=97.21ms  p(95)=105.26ms
iteration_duration...: avg=111.48ms min=60.15ms med=107.68ms max=199.57ms p(90)=131.64ms p(95)=145.17ms
iterations.........: 2696    89.632801/s
vus................: 10      min=10      max=10
vus_max............: 10      min=10      max=10
```

Figura 39 - Resultado completo do teste de listagem de 1.000 filmes utilizando gRPC

O resultado desse teste apresentou uma quantidade de 857 kB de dados enviados e 271 MB de dados recebidos, com um total de 2696 iterações, tempo médio de cada iteração de 111.48 milissegundos e tempo médio da requisição gRPC de 77.08 milissegundos.

Teste com 10.000 filmes

```
✓ status is OK

checks.............: 100.00% ✓ 234       ✗ 0
data_received.....: 437 MB  15 MB/s
data_sent.........: 211 kB  7.0 kB/s
grpc_req_duration....: avg=1.12s min=147.09ms med=1.1s  max=1.62s p(90)=1.38s p(95)=1.51s
iteration_duration...: avg=1.28s min=204.98ms med=1.25s max=1.79s p(90)=1.56s p(95)=1.68s
iterations.........: 234     7.748281/s
vus................: 10      min=10      max=10
vus_max............: 10      min=10      max=10
```

Figura 40 - Resultado completo do teste de listagem de 10.000 filmes utilizando gRPC

O resultado desse teste apresentou uma quantidade de 211 kB de dados enviados e 437 MB de dados recebidos, com um total de 234 iterações, tempo médio de cada iteração de 1.28 segundos e tempo médio da requisição gRPC de 1.12 segundos.

Teste com 150.000 filmes

```
✓ status is OK

checks.....................: 100.00% ✓ 20       ✗ 0
data_received..............: 995 MB  20 MB/s
data_sent..................: 59 kB   1.2 kB/s
grpc_req_duration..........: avg=22.05s min=20.17s med=22.16s max=22.76s p(90)=22.71s p(95)=22.73s
iteration_duration.........: avg=24.24s min=21.14s med=24.35s max=25.18s p(90)=25.16s p(95)=25.18s
iterations.................: 20      0.409427/s
vus........................: 9       min=9      max=10
vus_max....................: 10      min=10     max=10
```

Figura 41 - Resultado completo do teste de listagem de 150.000 filmes utilizando gRPC

O resultado desse teste apresentou uma quantidade de 59 kB de dados enviados e 995 MB de dados recebidos, com um total de 20 iterações, tempo médio de cada iteração de 24.24 segundos e tempo médio da requisição gRPC de 22.05 segundos.

Caso de uso de cadastro de filmes

REST

```
✓ success

checks.........................: 100.00% ✓ 17922     ✗ 0
data_received..................: 7.8 MB  259 kB/s
data_sent......................: 5.1 MB  171 kB/s
http_req_blocked...............: avg=4.05µs  min=0s     med=0s      max=6.09ms   p(90)=0s      p(95)=0s
http_req_connecting............: avg=30ns    min=0s     med=0s      max=540.5µs  p(90)=0s      p(95)=0s
http_req_duration..............: avg=16.6ms  min=9.72ms med=14.61ms max=123.84ms p(90)=23.85ms p(95)=27.79ms
  { expected_response:true }...: avg=16.6ms  min=9.72ms med=14.61ms max=123.84ms p(90)=23.85ms p(95)=27.79ms
http_req_failed................: 0.00%   ✓ 0         ✗ 17922
http_req_receiving.............: avg=46.97µs min=0s     med=0s      max=7.38ms   p(90)=0s      p(95)=504.69µs
http_req_sending...............: avg=6.77µs  min=0s     med=0s      max=2.87ms   p(90)=0s      p(95)=0s
http_req_tls_handshaking.......: avg=0s      min=0s     med=0s      max=0s       p(90)=0s      p(95)=0s
http_req_waiting...............: avg=16.54ms min=9.66ms med=14.57ms max=123.84ms p(90)=23.78ms p(95)=27.73ms
http_reqs......................: 17922   597.28211/s
iteration_duration.............: avg=16.73ms min=9.72ms med=14.72ms max=124.11ms p(90)=24.01ms p(95)=28.05ms
iterations.....................: 17922   597.28211/s
vus............................: 10      min=10      max=10
vus_max........................: 10      min=10      max=10
```

Figura 42 - Resultado completo do teste de cadastro de filmes utilizando REST

Com REST, o caso de uso para adicionar filmes foi executado 17922 vezes pelos 10 usuários virtuais durante 30 segundos, com 5.1 MB de dados enviados e 7.8 MB de dados recebidos. O tempo médio de cada iteração foi de

16.73 milissegundos, tempo médio de conexão HTTP foi de 30 nanossegundos, tempo médio de duração total da requisição HTTP foi de 16.6 milissegundos, tempo médio gasto para receber dados de resposta do servidor foi de 46.97 microssegundos, tempo médio gasto para enviar dados para o servidor foi de 6.77 microssegundos e tempo médio gasto aguardando a resposta do servidor foi de 16.54 milissegundos.

GraphQL

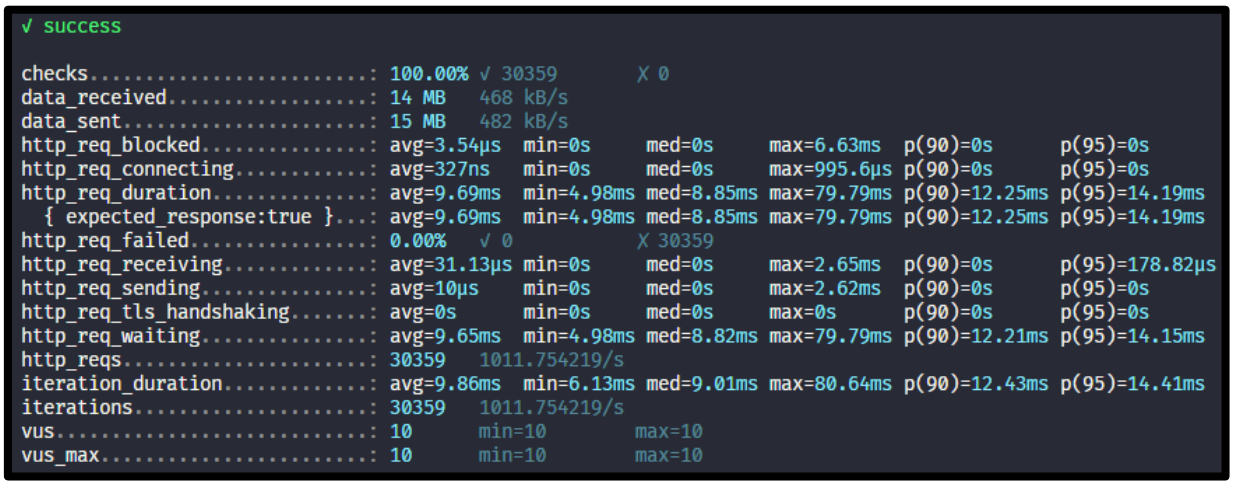


Figura 43 - Resultado completo do teste de cadastro de filmes utilizando GraphQL

Com GraphQL, o caso de uso para adicionar filmes foi executado 30359 vezes pelos 10 usuários virtuais durante 30 segundos, com 15 MB de dados enviados e 14 MB de dados recebidos. O tempo médio de cada iteração foi de 9.86 milissegundos, tempo médio de conexão HTTP foi de 327 nanossegundos, tempo médio de duração total da requisição HTTP foi de 9.69 milissegundos, tempo médio gasto para receber dados de resposta do servidor foi de 31.13 microssegundos, tempo médio gasto para enviar dados para o servidor foi de 10 microssegundos e tempo médio gasto aguardando a resposta do servidor foi de 9.65 milissegundos.

gRPC

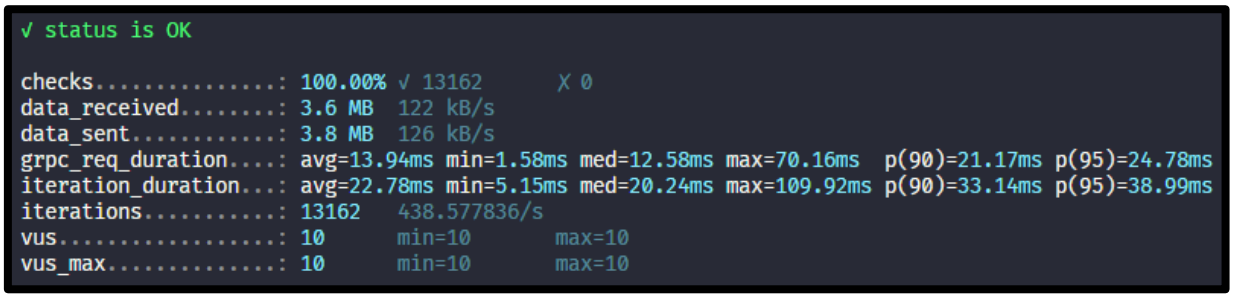


Figura 44 - Resultado completo do teste de cadastro de filmes utilizando gRPC

Com gRPC, o caso de uso para adicionar filmes foi executado 13162 vezes pelos 10 usuários virtuais durante 30 segundos, com 3.8 MB de dados enviados e 3.6 MB de dados recebidos. O tempo médio de cada iteração foi de 22.78 milissegundos e o tempo médio da requisição gRPC foi de 13.94 milissegundos.

Caso de uso de edição de filmes

REST

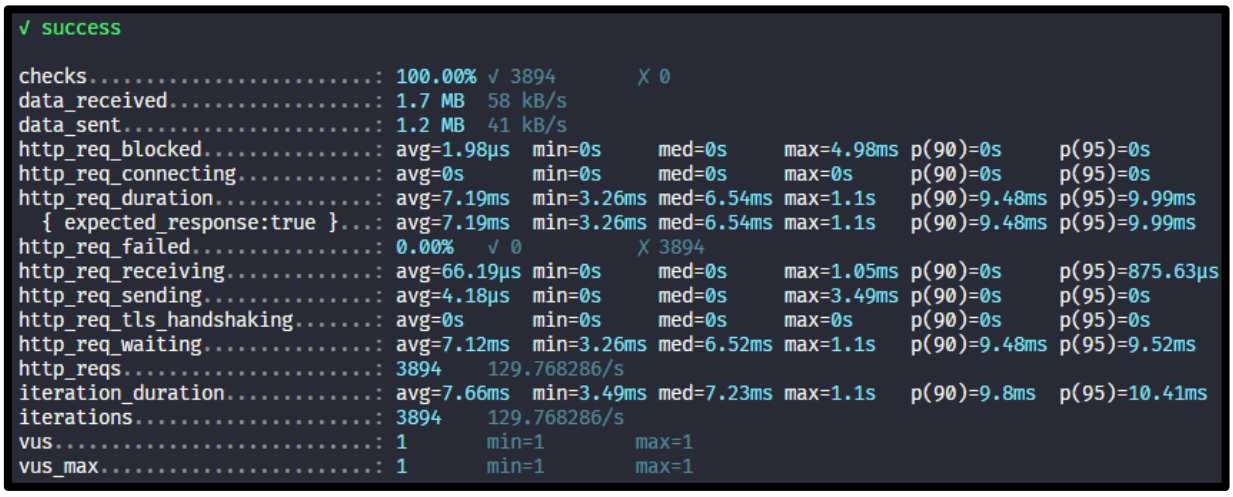


Figura 45 - Resultado completo do teste de edição de filmes utilizando REST

Com REST, o caso de uso para editar filmes foi executado 3894 vezes por 1 usuário virtual durante 30 segundos, com 1.2 MB de dados enviados e 1.7 MB de dados recebidos. O tempo médio de cada iteração foi de 7.66 milissegundos, tempo médio de conexão HTTP foi de 0 segundos, tempo médio de duração total da requisição HTTP foi de 7.19 milissegundos, tempo médio gasto para receber dados de resposta do servidor foi de 66.19 microssegundos, tempo médio gasto para enviar dados para o servidor foi de 4.18 microssegundos e tempo médio gasto aguardando a resposta do servidor foi de 7.12 milissegundos.

GraphQL

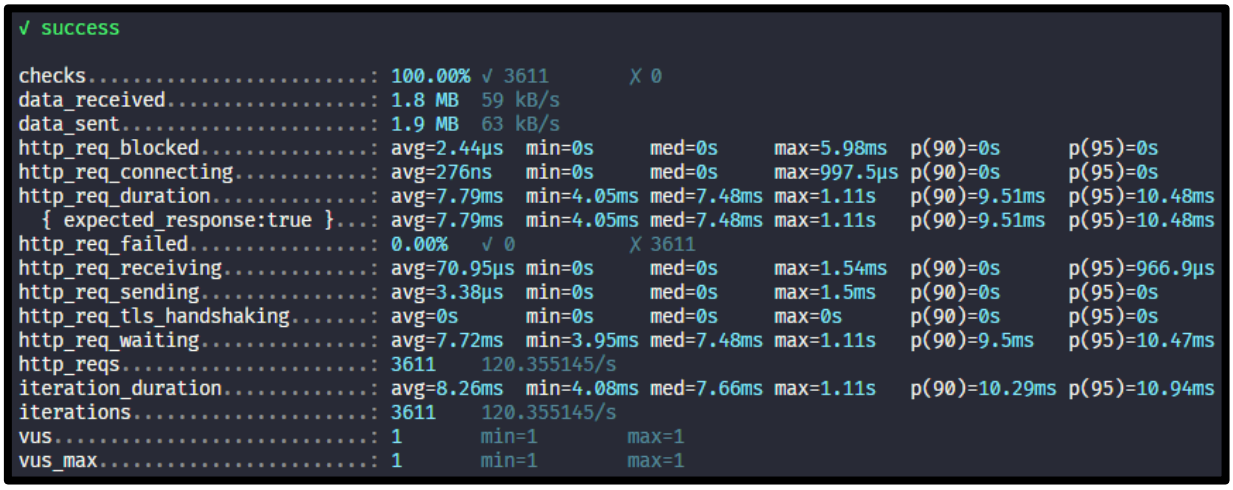


Figura 46 - Resultado completo do teste de edição de filmes utilizando GraphQL

Com GraphQL, o caso de uso para editar filmes foi executado 3611 vezes por 1 usuário virtual durante 30 segundos, com 1.9 MB de dados enviados e 1.8 MB de dados recebidos. O tempo médio de cada iteração foi de 8.26 milissegundos, tempo médio de conexão HTTP foi de 276 nanossegundos, tempo médio de duração total da requisição HTTP foi de 7.79 milissegundos, tempo médio gasto para receber dados de resposta do servidor foi de 70.95 microssegundos, tempo médio gasto para enviar dados para o servidor foi de 3.38 microssegundos e tempo médio gasto aguardando a resposta do servidor foi de 7.72 milissegundos.

gRPC

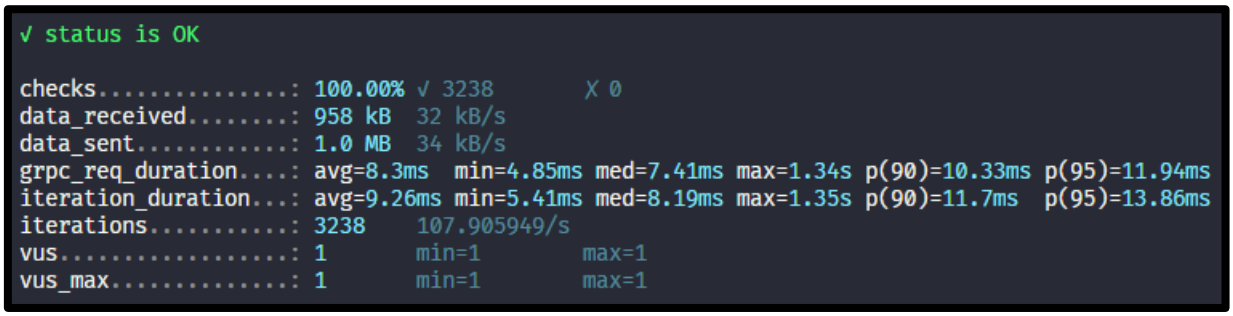


Figura 47 - Resultado completo do teste de edição de filmes utilizando gRPC

Com gRPC, o caso de uso para editar filmes foi executado 3238 vezes por 1 usuário virtual durante 30 segundos, com 1 MB de dados enviados e 958 kB de dados recebidos. O tempo médio de cada iteração foi de 9.26 milissegundos e o tempo médio da requisição gRPC foi de 8.3 milissegundos.

Caso de uso de exclusão de filmes

REST

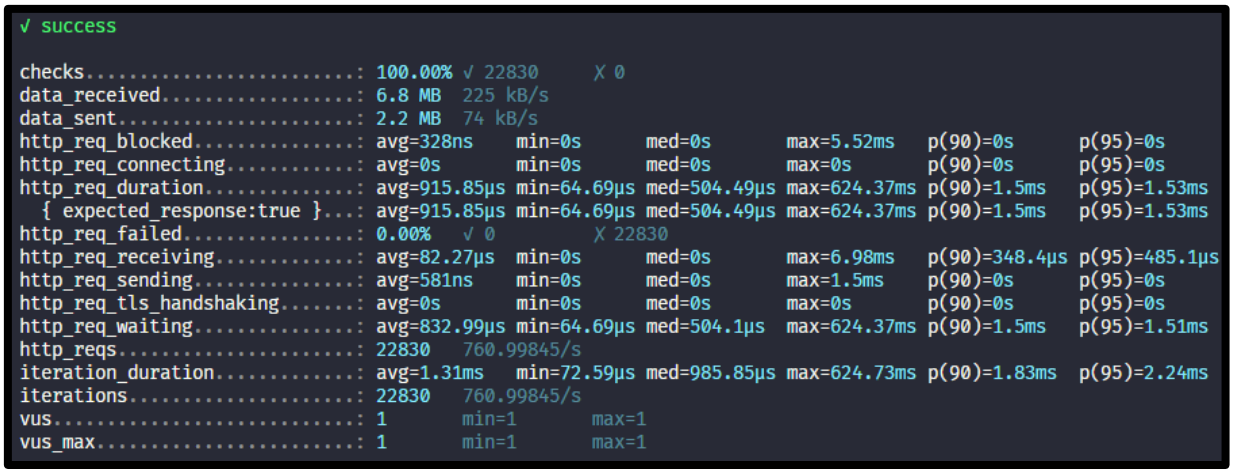


Figura 48 - Resultado completo do teste de exclusão de filmes utilizando REST

Com REST, o caso de uso para remover filmes foi executado 22.830 vezes por 1 usuário virtual durante 30 segundos, com 2.2 MB de dados enviados

e 6.8 MB de dados recebidos. O tempo médio de cada iteração foi de 1.31 milissegundos, tempo médio de conexão HTTP foi de 0 segundos, tempo médio de duração total da requisição HTTP foi de 915.85 microssegundos, tempo médio gasto para receber dados de resposta do servidor foi de 82.27 microssegundos, tempo médio gasto para enviar dados para o servidor foi de 581 nanossegundos e tempo médio gasto aguardando a resposta do servidor foi de 832.99 microssegundos.

GraphQL

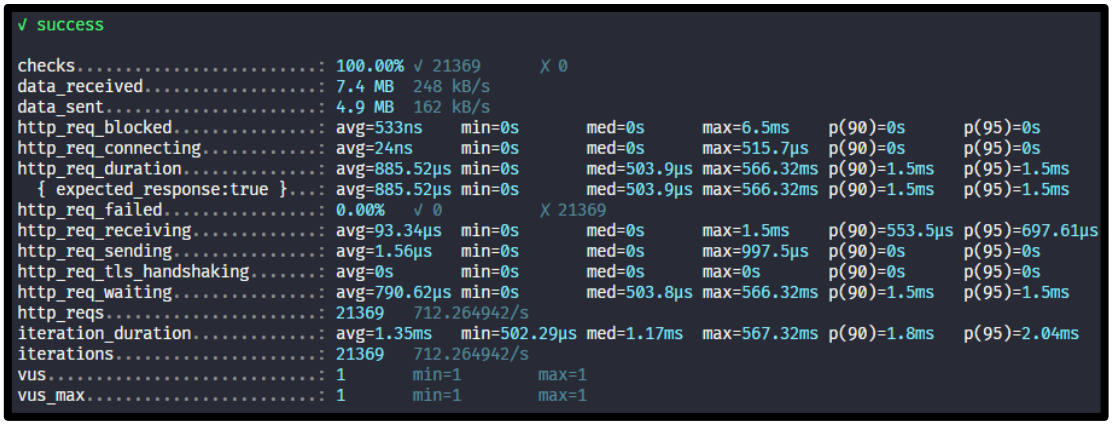


Figura 49 - Resultado completo do teste de exclusão de filmes utilizando GraphQL

Com GraphQL, o caso de uso para remover filmes foi executado 21.369 vezes por 1 usuário virtual durante 30 segundos, com 4.9 MB de dados enviados e 7.4 MB de dados recebidos. O tempo médio de cada iteração foi de 1.35 milissegundos, tempo médio de conexão HTTP foi de 24 nanossegundos, tempo médio de duração total da requisição HTTP foi de 885.52 microssegundos, tempo médio gasto para receber dados de resposta do servidor foi de 93.34 microssegundos, tempo médio gasto para enviar dados para o servidor foi de 1.56 microssegundos e o tempo médio gasto aguardando a resposta do servidor foi de 790.62 microssegundos.

gRPC

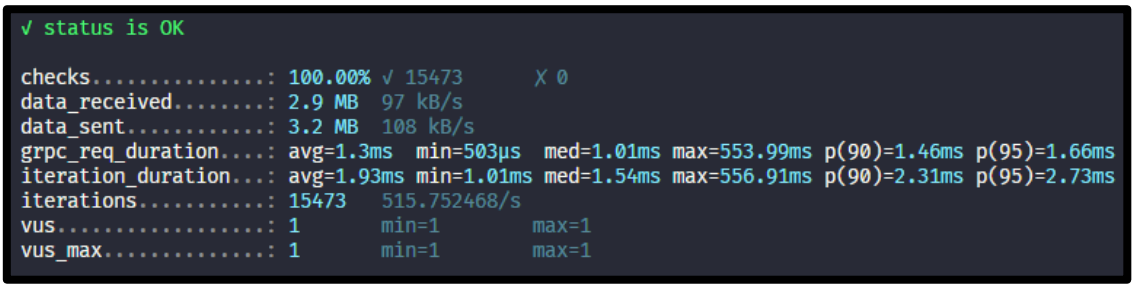


Figura 50 - Resultado completo do teste de exclusão de filmes utilizando gRPC

Com gRPC, o caso de uso para remover filmes foi executado 15.473 vezes por 1 usuário virtual durante 30 segundos, com 3.2 MB de dados enviados e 2.9 MB de dados recebidos. O tempo médio de cada iteração foi de 1.93 milissegundos e o tempo médio da requisição gRPC foi de 1.3 milissegundos.

APÊNDICE B – ARQUIVO PROTO DA SOLUÇÃO

Figura 51 - Arquivo proto com o serviço 'TccService' e com seus métodos e mensagens definidas

APÊNDICE C – CÓDIGO FONTE DA SOLUÇÃO

O código fonte da solução esta hospedado na plataforma GitHub em um repositório público, acessível pelo endereço https://github.com/NicolasMaso/rest-vs-graphql-vs-grpc.

APÊNDICE D – ARTIGO DO TRABALHO NO FORMATO SBC

Comparativo entre arquiteturas de APIs - REST, GraphQL e gRPC

Nicolas Nascimento Maso[1], José Eduardo De Lucca[2]

[1]Departamento de Informática e Estatística (INE)
Universidade Federal de Santa Catarina (UFSC) – Florianópolis, SC – Brazil

nicolas.maso@grad.ufsc.br, jose.lucca@ufsc.br

Abstract. With the growth of the technology market, new ideas and solutions emerge every day to solve problems. Despite the technologies available, many projects run into problems or are terminated prematurely due to poor decisions. This increases the demand for qualified professionals to create and maintain these solutions. New technologies and updates emerge constantly, while others become obsolete. This paper carries out a comparative analysis of three API architectures (REST, GraphQL and gRPC), highlighting their characteristics, documentation, performance and other topics, helping developers and architects to make better decisions in their projects.

Resumo. Com o crescimento do mercado de tecnologia, novas ideias e soluções surgem diariamente para resolver problemas. Apesar das tecnologias disponíveis, muitos projetos enfrentam problemas ou são encerrados prematuramente devido a decisões inadequadas. Isso aumenta a demanda por profissionais qualificados para criar e manter essas soluções. Novas tecnologias e atualizações surgem constantemente, enquanto outras se tornam obsoletas. Este trabalho realiza uma análise comparativa entre três arquiteturas de APIs (REST, GraphQL e gRPC), destacando suas características, documentação, desempenho e outros tópicos, ajudando desenvolvedores e arquitetos a tomarem melhores decisões em seus projetos.

1. Introdução

Nos últimos anos o setor de TIC (Tecnologia da informação e comunicação) gerou milhares de empregos tanto a nível nacional como a nível mundial, e além disso, o setor possui previsão de muita demanda para os próximos anos e possui uma remuneração superior em comparação com outros setores. Segundo os últimos estudos realizados pela Brasscom (2021), que é a Associação das Empresas de Tecnologia da Informação e Comunicação e de Tecnologias Digitais, apenas no Brasil no ano de 2021, o macrossetor gerou mais de 198 mil empregos e em 2022 mais de 117 mil empregos. Nesse sentido, a área de desenvolvimento de software está com uma alta demanda de profissionais de tecnologia qualificados como arquitetos e desenvolvedores de software no mercado mundial, e, para se tornar um profissional qualificado, é necessário aprender e saber usufruir das excelentes tecnologias disponíveis que temos atualmente.

No cenário de desenvolvimento de software, um dos conceitos mais citados é o de Application Programming Interface (API), que em português significa Interface de Programação de Aplicação. De forma resumida, uma API é um conjunto de rotinas e padrões que permitem a comunicação entre dois ou mais softwares, ou seja, fornece uma maneira padronizada para que sistemas acessem dados e funcionalidades de outros sistemas. As APIs desempenham um papel fundamental na integração de sistemas e no desenvolvimento de novas aplicações, e a escolha de uma arquitetura pode ter um impacto significativo na eficiência e no desempenho das soluções tecnológicas, podendo inclusive ter impacto na economia de recursos.

As arquiteturas de APIs Representational State Transfer (REST), Graph Query Language (GraphQL) e gRPC Remote Procedure Calls (gRPC) foram selecionadas para o desenvolvimento do presente estudo devido a sua popularidade na comunidade de desenvolvimento de software, que se deram principalmente devido a popularização da arquitetura de microsserviços, a qual abriu caminho para novas opções de comunicação entre aplicações.

Com isso, o objetivo principal do trabalho foi realizar uma análise e comparação entre três das principais arquiteturas de desenvolvimento de APIs utilizadas no mercado atualmente, sendo elas as arquiteturas REST, GraphQL e gRPC, em relação aos aspectos de facilidade de implementação, qualidade da documentação, relevância na comunidade de software e o desempenho baseado em métricas específicas.

2. Arquiteturas

2.1 REST

A arquitetura REST (Representational State Transfer ou Transferência de Estado Representacional) é um estilo arquitetônico para desenvolvimento de APIs que foi criado pelo cientista da computação norte-americano Roy Thomas Fielding (2000) durante sua tese de doutorado no ano de 2000. Por se tratar de um modelo arquitetural, REST independe de uma linguagem de programação. Nesse sentido, é possível implementar uma API na arquitetura REST em diversas linguagens de programação. Segundo Fielding (2000), para que uma API seja desenvolvida na arquitetura REST é necessário que ela atenda as seguintes restrições:

- Possua um estilo arquitetônico cliente-servidor
- Não possua estado
- Realize armazenamento em cache
- Possua uma interface uniforme
- Seja desenvolvido como um sistema em camadas
- Gere código sob demanda

O detalhamento de cada restrição está detalhado em sua tese (FIELDING, 2000) e também na dissertação referenciada no presente artigo. A ideia principal do modelo arquitetural REST é a abstração de informações em recursos que, segundo o autor, é qualquer informação que possa ser nomeada, como um documento, uma imagem, um serviço, uma coleção de outros recursos, um objeto, uma pessoa e assim por diante. Esses recursos são criados, acessados e modificados por meio de requisições e respostas,

realizadas respectivamente pelo cliente e servidor. Para que um recurso possa ser identificado em uma comunicação via API REST, é preciso que cada recurso possua seu identificador, pois é o identificador do recurso que irá possibilitar a referência ao mesmo. A comunicação com REST é baseada no protocolo HTTP, que é o principal protocolo responsável pela transferência de dados na internet, nesse sentido, as requisições são executadas com base nos métodos HTTP, como POST, PUT, GET, DELETE, e demais métodos especificados no protocolo. As respostas do servidor podem ser retornadas em diversos formatos como JavaScript Object Notation (JSON), HTML, Extensible Markup Language (XML), texto, entre outros.

Tabela 1 – Exemplo de possíveis rotas em uma API REST

Rota (recurso)	**Método HTTP**	**Ação**
/usuarios	POST	Cadastro de um usuário
/usuarios	GET	Listagem de usuários
/usuarios/{id}	PUT	Atualização de um usuário
/usuarios/{id}	DELETE	Exclusão de um usuário

Exemplificando, um possível recurso em uma API REST poderia ser 'usuários' para tratar ações na API que fossem relacionadas somente a usuários, seja um cadastro, atualização, listagem ou exclusão. Nesse exemplo, de acordo com a ação deseja pelo cliente, bastaria uma solicitação indicando a rota juntamente com um método HTTP e aguardar o retorno do servidor. Considerando a atualização de um usuário, bastaria acessar o endereço da api informando a rota, indicar o método HTTP PUT e informar os parâmetros necessários para realizar a ação desejada, como por exemplo *https://dominio.com/api/usuarios/1*, aonde '**usuarios'** seria o recurso e o valor '**1**' seria o parâmetro indicando o identificador do usuário a ser atualizado.

O fluxo geral da comunicação de uma API REST ocorre da seguinte forma:

1) O cliente envia uma solicitação HTTP ao servidor com um método, recurso, parâmetro e tudo que for necessário de acordo com a documentação da API para formatar a solicitação de modo que o servidor entenda.
2) Caso a solicitação esteja de acordo com o que o servidor espera, a requisição é recebida e processada pelo servidor.
3) Por fim o servidor retorna uma resposta ao cliente com informações que indicam se a requisição foi realizada com sucesso ou não, e com as informações que podem ter sido solicitadas pelo cliente.

2.2 GraphQL

GraphQL é uma linguagem de consulta e um mecanismo de execução que foi originalmente criado pela empresa Facebook no ano de 2012 para descrever os recursos e os requisitos dos modelos de dados para aplicações no estilo cliente-servidor (GRAPHQL, 2024a). Em 2015 foi iniciado o processo para tornar a linguagem open standard, que significa padrão aberto em português, e a partir desse ano a linguagem passou a ser de livre acesso e implementação.

Conforme dito anteriormente, GraphQL é uma linguagem e um mecanismo de execução do lado do servidor, que possibilita a execução de consultas usando um sistema de tipos que é definido pelo próprio desenvolvedor, ou seja, não se trata de um estilo arquitetural como REST, mas sim uma linguagem que possui sua especificação e sintaxe específica (GRAPHQL, 2024b). A linguagem GraphQL não é atrelada a nenhum banco de dados ou mecanismo de armazenamento específico, é apenas vinculada a seu código e dados existentes. Além disso, um serviço GraphQL pode ser desenvolvido com qualquer framework backend ou linguagem de programação (GRAPHQL, 2024b).

De forma resumida, a linguagem GraphQL trata basicamente da seleção de campos em objetos. Então, para que um serviço GraphQL seja criado, é preciso que seja definido um schema, que é um conjunto de tipos (objetos) que possuem campos (propriedades), e, associado a esse schema, também devem ser definidas funções para acesso a cada campo de cada tipo definido. Tendo em vista isso, existem três tipos desse conjunto que são especiais ou também conhecidos como root (raiz/base), o tipo “Query”, o tipo “Mutation” e o tipo “Subscription”, e servem respectivamente para as operações de consultas, mutações e assinaturas, tais operações serão descritas no próximo parágrafo. Além disso, o primeiro tipo (query) é obrigatório e os demais são opcionais para a execução de um serviço GraphQL.

Apesar de que em sua definição conste “linguagem de consulta”, assim como uma API desenvolvida na arquitetura REST, uma API GraphQL, além de buscar dados também possibilita a manipulação e criação de dados. Tendo em vista isso, temos em GraphQL o tipo de operação, que pode ser “query” (consulta), “mutation” (mutação) ou “subscription” (assinatura), as quais servem, respectivamente, para uma busca somente de leitura, uma gravação seguida de uma busca e uma solicitação de longa duração que busca dados em resposta a eventos de origem (GRAPHQL, 2024a).

Após o serviço GraphQL iniciar sua execução, que geralmente ocorre por meio de algum Uniform Resource Locator (URL), que é basicamente uma sequência de texto que especifica onde um recurso pode ser encontrado na Internet (MDN WEB DOCS, 2024b), ele pode receber consultas para validação e execução. Ao receber uma consulta, o serviço primeiramente verifica se a consulta está solicitando tipos e campos que foram definidos e na sequência executa as funções fornecidas para gerar um resultado (GRAPHQL, 2024b). Assim como em uma API desenvolvida em REST é possível passar argumentos (parâmetros com valor fixo) ou variáveis (parâmetros com valor variável) para uma consulta. A diferença é que com GraphQL é possível passar um conjunto de argumentos ou variáveis para cada campo e objeto aninhado, sendo que em REST cada requisição recebe apenas um conjunto de argumentos ou variáveis. De forma resumida, uma API GraphQL funciona com a definição de dados no lado do servidor e com isso o lado cliente consegue solicitar, criar e alterar somente o que deseja, gerando então resultados mais previsíveis e uma maior flexibilidade do lado do cliente, conforme exemplo da Figura 1.

Figura 1. Exemplo de consulta GraphQL

2.3 gRPC

Segundo o site oficial, o gRPC (gRPC Remote Procedure Calls) foi criado inicialmente pela empresa Google, que há mais de uma década utilizava uma única infraestrutura de RPC de uso geral chamada Stubby para conectar o grande número de microsserviços executados dentro e entre seus data centers (GRPC, 2024b). A partir disso, em março de 2015, a Google decidiu criar uma nova versão do Stubby e torná-la de código aberto, o que resultou na criação do gRPC. Apesar de ter sido criada para utilização interna da empresa, o gRPC pode ser de código aberto já é utilizado por diversas organizações como Netflix, Square, Cisco, entre outras (GRPC, 2024b). Ainda segundo o site oficial, o gRPC é conceituado como uma estrutura moderna de RPC (Remote Procedure Call ou Chamada de Procedimento Remoto) de código aberto e que pode ser executada em qualquer ambiente, servindo para diversos casos de uso, desde a conexão entre serviços como também na última milha da computação, servindo para conectar dispositivos, aplicativos móveis e navegadores a serviços back-end (GRPC, 2024a).

Como consta no próprio nome, a implementação do gRPC é baseada em RPC, que é um protocolo que fornece o paradigma de comunicações de alto nível usado no sistema operacional (IBM, 2024). O protocolo RPC pressupõe a existência de um protocolo de transporte de baixo nível, como o Protocolo de Controle de Transmissão/Protocolo de Internet (TCP/IP) ou o Protocolo de Datagrama do Usuário (UDP), sendo o último o mais utilizado geralmente, para transportar os dados da mensagem entre os programas de comunicação. O RPC implementa um sistema lógico de comunicação cliente-servidor projetado especificamente para o suporte de aplicativos de rede (IBM, 2024). Nesse sentido, com gRPC, uma aplicação cliente pode chamar um método diretamente em uma aplicação servidor em uma máquina diferente, como se fosse um objeto local, o que consequentemente facilita a criação de aplicativos e serviços distribuídos (GRPC, 2024c). Assim como em muitos sistemas que utilizam RPC, o gRPC baseia-se na ideia de definir um serviço, especificando os métodos que podem ser chamados remotamente com seus parâmetros e tipos de retorno. No lado do servidor, o servidor implementa essa interface e executa um servidor gRPC para lidar com as chamadas do cliente. No lado do cliente, o cliente tem um stub (chamado apenas de cliente em algumas linguagens) que fornece os mesmos métodos que o servidor (GRPC, 2024c).

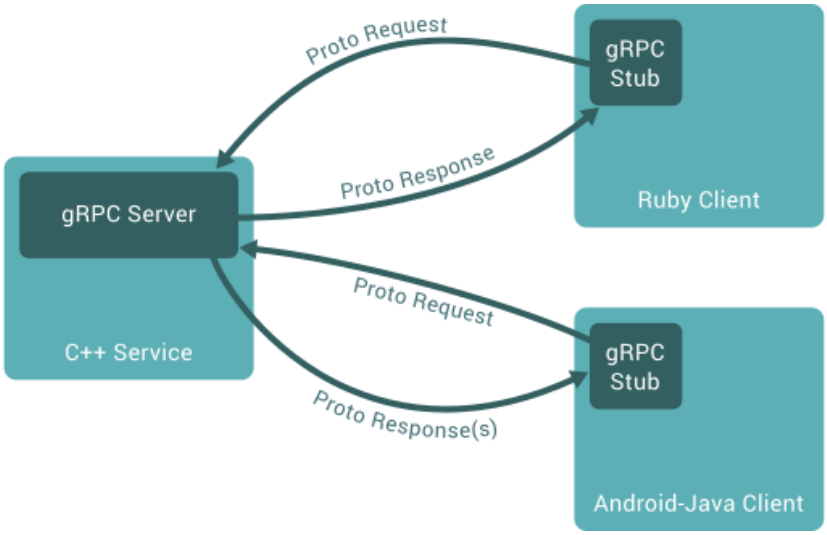


Figura 2. Exemplo de comunicação utilizando gRPC

Ainda segundo o site oficial (GRPC, 2024c), os clientes e servidores gRPC podem ser executados e se comunicarem em vários tipos de ambientes, além de ter suporte a diversas linguagens. Dessa maneira, conforme exemplo da Figura 2, é possível criar um servidor gRPC utilizando uma linguagem como C++ e utilizar clientes com outras linguagens como Ruby ou Java para consumir o serviço normalmente.

Por padrão, um servidor gRPC utiliza ‘Protocol Buffers’ (também conhecido como protobuf ou em português buffers de protocolo) como linguagem de definição de interface, que é uma linguagem para serialização de dados estruturados de código aberto que também foi criado pela Google, mas que não impede que possa ser utilizado outros formatos de dados como JSON (GRPC, 2024c).

Para ser possível a comunicação com buffers de protocolo, é necessário primeiramente definir a estrutura dos dados que será serializada em um arquivo proto, que é basicamente um arquivo de texto comum, mas que possui a extensão ‘.proto’. Os dados de um buffer de protocolo são estruturados como mensagens, sendo que cada mensagem é um pequeno registro lógico de informações que contém uma série de pares nome-valor que são chamados de campos (GRPC, 2024c).

```
message Person {
  string name = 1;
  int32 id = 2;
  bool has_job = 3;
}
```

Figura 3. Exemplo de uma mensagem definida em um arquivo proto

Na Figura 3, temos como exemplo uma mensagem chamada **'*Person*'** que possui os campos '***name***', '***id***' e '***has*_job'** com seus respectivos tipos e números identificadores. Segundo documentação oficial da linguagem 'Protocol Buffers' (LANGUAGE…, 2024), os números identificadores são obrigatórios e únicos em cada mensagem e servem para o processo de codificação dos dados.

Também em arquivo proto, é necessário definir os serviços gRPC com seus respectivos parâmetros de método RPC e seus tipos de parâmetros de entrada e de retorno que devem ser especificados como mensagens de protocol buffer. Com a definição de um serviço gRPC em um arquivo proto, o gRPC por meio do *Protocol Buffer Compiler Installation*, fornece plugins de compilador de buffer de protocolo que geram código tanto do lado do cliente como do lado do servidor. Após isso, a utilização desse código gerado é utilizado para implementar a API correspondente no lado do servidor e consequentemente para que a API do servidor seja utilizada pelo lado do cliente. Dessa forma, o servidor implementa os métodos do serviço que foram declarados no arquivo proto e executa um servidor gRPC para responder às chamadas do cliente. A infraestrutura gRPC decodifica as solicitações recebidas, executa os métodos do serviço e codifica as respostas do serviço. No lado do cliente há um objeto local conhecido como stub (em certas linguagens o termo preferido é cliente) que implementa os mesmos métodos do serviço, dessa forma, o cliente pode chamar os métodos no objeto local que envolvem os parâmetros da chamada no tipo de mensagem de buffer de protocolo correspondente, envia essas solicitações ao servidor e retorna as respostas de buffer de protocolo do servidor (GRPC, 2024d).

3. Comparativo

Uma das principais diferenças entre as arquiteturas do estudo é a versão do protocolo HTTP utilizada para a comunicação com a API. O protocolo HTTP ou protocolo de transferência de hipertexto serve para buscar recursos, como por exemplo um documento HTML, e é a base de qualquer troca de dados na Web (MDN WEB DOCS, 2024a). Além disso é um protocolo cliente-servidor, o que significa que as solicitações são iniciadas pelo destinatário, que no cenário da Web geralmente é um navegador. Conforme a tabela anterior, tanto REST como GraphQL fazem uso da versão 1.1 do protocolo HTTP, que foi a primeira versão utilizável do protocolo lançada no ano de 1997. Já as APIs desenvolvidas com gRPC utilizam a versão 2 do protocolo HTTP que foi lançada oficialmente no ano de 2015 (MDN WEB DOCS, 2024a).

Tabela 2 – Comparativo entre as arquiteturas REST, GraphQL e gRPC

Arquitetura	REST	GraphQL	gRPC
Protocolo HTTP	HTTP/1.1	HTTP/1.1	HTTP/2
Formato	• JSON • XML • Texto • Outros	• JSON • Outros	• Buffers de protocolo • JSON
Comunicação	Solicitação de cliente	Solicitação de cliente	• Solicitação de cliente • Fluxo contínuo
Implementação	Rápida	Rápida	Lenta
Abordagem	Recursos	Consultas	Serviços
Geração de código	Terceiros	Integrado	Integrado
Tratamento de erros	Fraco	Médio	Médio
Autenticação e autorização	Terceiros	Terceiros	• Integrado (SSL/TLS, ALTS e token com Google) • Terceiros
Tipagem	Fraca	Forte	Forte

Com base no estudo de cada arquitetura, é notável que cada uma possui casos de uso mais adequados em relação às demais, tendo influência ou não de sua origem. Por se tratar de uma arquitetura, é possível aplicar REST em qualquer cenário, já GraphQL pode trazer um maior controle do cliente nas solicitações realizadas, pode funcionar melhor em cenários onde existem consultas mais complexas ou diversificadas. O framework gRPC foi concebido inicialmente para comunicação entre serviços de um data center e também entre outros data centers, o que faz dele uma ótima opção para cenários como esses e também cenários de sistemas distribuídos e microsserviços. Apesar das características de cada uma, atualmente existem diversos *frameworks*, bibliotecas e softwares de terceiros que auxiliam na implementação de uma API, seja em REST, GraphQL ou gRPC

4. Desenvolvimento

Apesar de já existirem estudos que demonstram o desempenho entre as arquiteturas de APIs REST, GraphQL e gRPC, nem todos os estudos podem ser utilizados inteiramente como fonte de informação e escolha para uma tomada de decisão, visto que na área de desenvolvimento de software existem muitos fatores que podem influenciar o desempenho de uma aplicação, como a forma que ela foi desenvolvida, a conexão com a internet, a performance do computador onde a aplicação está hospedada, a linguagem de programação utilizada, e, a forma que foi implementada. Dessa forma, a solução proposta do presente trabalho é gerar mais um estudo para a literatura, porém com a aplicação de boas práticas de desenvolvimento e uma boa metodologia, para que possa ser útil e de confiabilidade para a comunidade de desenvolvimento de software. O principal objetivo é realizar um estudo das arquiteturas de APIs REST, GraphQL e gRPC e desenvolver uma API que contemple as mesmas, que permita a discussão e análise das características, limitações, pontos positivos e negativos e o desempenho geral de cada uma utilizando a ferramenta para testes k6 da empresa Grafana para obter as métricas gerais de quantidade de dados recebidos, quantidade de dados enviados e tempo para concluir uma iteração completa, além de métricas específicas para o protocolo HTTP, sendo elas, tempo total de uma requisição, taxa de solicitações com falha, tempo gasto para estabelecer a conexão TCP, tempo gasto no envio de dados, tempo gasto para receber dados de resposta e total de solicitações HTTP, bem como métricas específicas para a arquitetura gRPC, com o tempo para receber uma resposta, número total de fluxos iniciados, número total de mensagens recebidas e número total de mensagens enviadas.

A API foi desenvolvida com uma arquitetura que permite que um mesmo caso de uso seja executado via REST, GraphQL ou gRPC, ou seja, o que diferencia é a porta de entrada para acessar o caso de uso na API e o retorno dos dados, seja com qualquer uma das tecnologias mencionadas. Além disso, a API foi desenvolvida com baixo acoplamento, utilizando os princípios SOLID e outros princípios da arquitetura limpa, que são princípios de design de código e que foram apresentados pela primeira vez pelo famoso cientista da computação Robert C. Martin no seu livro Arquitetura Limpa (MARTIN, 2019).

5. Testes

Foi utilizado a ferramenta k6 para os testes, essa ferramenta possui métricas gerais que são geradas sempre para qualquer protocolo que é testado e também métricas específicas para alguns protocolos, como é o caso do gRPC que foi utilizado no presente estudo.

Todos os testes de criação e listagem foram realizados com uma configuração para simular 10 usuários virtuais e por um tempo de utilização de 30 segundos, já os testes de edição e remoção foram realizados com uma configuração para simular 1 usuário virtual por um tempo de utilização de 30 segundos, pois devido a necessidade de enviar identificadores como parâmetro, não foi possível configurar para que usuários virtuais diferentes enviassem identificadores diferentes. Além disso, todos os testes de listagem foram realizados com quatro quantidades de itens fixas, sendo respectivamente 1.000, 10.000, 150.000 e 400.000 itens, exceto com gRPC que no cenário com 400.000 itens não foi possível executar devido ao limite de dados recebidos em bytes pela ferramenta de teste. Os casos de uso selecionados para a realização dos testes foram ‘ListMovies’

(listagem de filmes), ‘AddMovie’ (cadastro de um filme), ‘EditMovie’ (edição de dados de um filme) e ‘DeleteMovie’ (exclusão de um filme).

Em todos os cenários de testes que foram executados, os testes foram repetidos múltiplas vezes para gerar uma maior confiabilidade nos resultados do estudo. Para o presente artigo os resultados foram apresentados com apenas três métricas, sendo o número total de iterações, o tempo médio de cada iteração e o tempo médio aguardando a resposta do servidor, pois essas métricas são melhores para um comparativo geral de desempenho.

Caso de uso de listagem de filmes

O caso de uso ‘**ListMovies**’ realiza uma busca de filmes no banco de dados de acordo com a quantidade passada por um parâmetro numérico chamado ‘**limit**’, exemplificando, caso seja informado o valor 1000 para o respectivo parâmetro, o retorno contará com dados de 1000 filmes. As Tabelas 3, 4, 5 e 6 mostram os resultados obtidos nesse caso de uso.

Tabela 3 - Comparativo de desempenho do caso de uso 'ListMovies' com 1.000 itens

Quantidade de dados	Arquitetura	Total de iterações	Tempo médio de cada iteração	Tempo médio aguardando resposta do servidor
1.000	REST	2651	113.36 ms	112.7 ms
1.000	GraphQL	1375	218.93 ms	218.25 ms
1.000	gRPC	2696	111.48 ms	77.08 ms

Tabela 4 - Comparativo de desempenho do caso de uso 'ListMovies' com 10.000 itens

Quantidade de dados	Arquitetura	Total de iterações	Tempo médio de cada iteração	Tempo médio aguardando resposta do servidor
10.000	REST	238	1.28 s	1.28 s
10.000	GraphQL	139	2.24 s	2.23 s
10.000	gRPC	234	1.28 s	1.12 s

Tabela 5 - Comparativo de desempenho do caso de uso 'ListMovies' com 150.000 itens

Quantidade de dados	Arquitetura	Total de iterações	Tempo médio de cada iteração	Tempo médio aguardando resposta do servidor
150.000	REST	21	17.02 s	16.95 s
150.000	GraphQL	17	26.27 s	26.19 s
150.000	gRPC	20	24.24 s	22.05 s

Tabela 6 - Comparativo de desempenho do caso de uso 'ListMovies' com 400.000 itens

Quantidade de dados	Arquitetura	Total de iterações	Tempo médio de cada iteração	Tempo médio aguardando resposta do servidor
400.000	REST	8	44.2 s	43.87 s
400.000	GraphQL	4	45.08 s	44.75 s
400.000	gRPC	--	--	--

Com base nos quatro cenários de teste para o caso de uso de listagem de filmes, como resultado REST obteve um desempenho superior com 10.000, 150.000 e 400.000 itens retornados. Já o gRPC obteve o melhor desempenho no cenário com 1.000 itens retornados.

Caso de uso de cadastro de filmes

O caso de uso '**AddMovie**' é responsável pelo cadastro de um filme e recebe como parâmetro os dados necessários para cadastrar um filme, cria o objeto de um filme e faz o registro no banco de dados. A Tabela 7 mostra os resultados obtidos nesse caso de uso.

Tabela 7 - Comparativo de desempenho do caso de uso 'AddMovie'

Arquitetura	Total de iterações	Tempo médio de cada iteração	Tempo médio aguardando resposta do servidor
REST	17.922	16.73 ms	16.54 ms
GraphQL	30.359	9.86 ms	9.65 ms
gRPC	13.162	22.78 ms	13.94 ms

No caso de uso de cadastro de filmes, GraphQL obteve um desempenho bastante superior em relação a REST e gRPC, com praticamente o dobro de inserções.

Caso de uso de edição de filmes

O caso de uso '**EditMovie**' é responsável por editar dados de um filme e recebe como parâmetro o identificador do filme e também os dados do filme que serão atualizados. É realizado uma busca do filme existente no banco de dados pelo identificador recebido como parâmetro e na sequência os dados do filme são atualizados no banco de dados. A Tabela 8 mostra os resultados obtidos nesse caso de uso.

Tabela 8 - Comparativo de desempenho do caso de uso 'EditMovie'

Arquitetura	Total de iterações	Tempo médio de cada iteração	Tempo médio aguardando resposta do servidor
REST	3.894	7.66 ms	7.12 ms
GraphQL	3.611	8.26 ms	7.72 ms
gRPC	3.238	9.26 ms	8.30 ms

Os resultados obtidos no caso de uso de edição de um filme demonstraram um maior desempenho com a arquitetura REST, porém tanto GraphQL como gRPC apresentaram um desempenho próximo a REST.

Caso de uso de exclusão de filmes

O caso de uso '**DeleteMovie**' é responsável por excluir um filme e recebe como parâmetro o identificador de um filme. Com base no identificador recebido, o filme é removido do banco de dados. A Tabela 9 mostra os resultados obtidos nesse caso de uso.

Tabela 9 - Comparativo de desempenho do caso de uso 'DeleteMovie'

Arquitetura	Total de iterações	Tempo médio de cada iteração	Tempo médio aguardando resposta do servidor
REST	22.830	1.31 ms	832.99 µs
GraphQL	21.369	1.35 ms	790.62 µs
gRPC	15.473	1.93 ms	1.3 ms

No caso de uso de exclusão de filmes, REST obteve um melhor desempenho e GraphQL obteve um desempenho bastante próximo. A arquitetura gRPC ficou um pouco distante em relação aos demais nesse cenário de teste.

5.1 Comparativo dos testes

Todos os testes foram realizados com sucesso, exceto a listagem com 400.000 itens que não foi possível realizar com gRPC devido a uma limitação de recebimento de dados pela ferramenta de testes, mesmo consultando a documentação não foi possível solucionar e por isso para o cenário de teste de listagem com a maior quantidade de dados com gRPC foi possível listar somente 150.000 itens, o que excluiu a arquitetura da comparação com REST e GraphQL nesse cenário.

Em relação ao cadastro de itens, GraphQL obteve o melhor resultado com uma média de 9.86 milissegundos por iteração e realizando 30.359 cadastros em um período de 30 segundos, REST ficou em segundo lugar com 17.922 cadastros realizados e uma média de 16.73 milissegundos por iteração e gRPC em terceiro lugar com 13.162 cadastros e uma média de 22.78 milissegundos por iteração.

Em relação a edição de itens, REST obteve o melhor desempenho com 3.894 iterações e uma média de 7.66 milissegundos, GraphQL na sequência obteve 3.611 iterações e uma média de 8.26 milissegundos e gRPC obteve 3.238 iterações com uma média de 9.26 milissegundos.

Em relação a exclusão de itens, REST também obteve o melhor desempenho com 22.830 iterações e uma média de 1.31 milissegundos, GraphQL ficou muito próximo na

sequência com 21.369 iterações e uma média de 1.35 milissegundos e gRPC obteve 15.473 iterações com uma média de 1.93 milissegundos.

Por fim, em relação a listagem de itens, no cenário com 1.000 itens o gRPC obteve melhor resultado com 2.696 iterações e uma média de 111.48 milissegundos por iteração, REST obteve o segundo melhor resultado com 2.651 iterações e uma média de 113.35 milissegundos por iteração e GraphQL obteve o terceiro lugar com 1.375 iterações e uma média de 218.93 milissegundos por iteração. No cenário com 10.000 itens, REST obteve o melhor desempenho com 238 iterações e uma média por iteração de 1.28 segundos, gRPC ficou muito próximo com 234 iterações e também uma média por iteração de 1.28 segundos e GraphQL novamente obteve o terceiro lugar com 139 iterações e uma média por iteração de 2.24 segundos. No terceiro cenário com 150.000 itens, REST obteve o melhor desempenho com 21 iterações e uma média de 17.02 segundos por iteração, gRPC ficou muito próximo e obteve um resultado de 20 iterações com uma média por iteração de 24.24 segundos e GraphQL ficou em terceiro lugar, mas com um resultado bem próximo com 17 iterações e uma média de 26.27 segundos por iteração. Na disputa do último cenário entre REST e GraphQL com 400.000 itens, REST obteve o melhor resultado com 8 iterações e uma média por iteração de 44.2 segundos contra 4 iterações e uma média de 45.08 segundos da linguagem GraphQL.

Com base nos resultados, temos que REST obteve o melhor desempenho no geral pois realizou mais exclusões, edições e também ganhou em 3 dos 4 cenários de teste de listagem. Na sequência temos a linguagem GraphQL que obteve o melhor desempenho em relação ao cadastro, com uma diferença grande de 12.437 itens a mais em relação a REST e 17.197 itens a mais em relação ao gRPC, além de ser superior a gRPC nas edições e exclusões, tendo o pior desempenho apenas nas listagens. Apesar de gRPC obter o terceiro melhor desempenho no cenário geral, vale destacar que em relação às listagens, gRPC obteve o melhor desempenho com 1.000 itens e praticamente empatou com REST nos cenários de 10.000 e 150.000 itens com a mesma média de tempo por iteração e 4 iterações de diferença no total no cenário de 10.000 itens e apenas uma iteração de diferença no cenário com 150.000 itens. Conforme já destacado na presente seção, devido a limitação de dados não foi possível executar o cenário de teste com 400.000 itens com gRPC.

6. Conclusão

Por meio deste trabalho, foi possível entender o funcionamento geral de cada arquitetura em estudo, com suas respectivas características, modo de funcionamento, vantagens, desvantagens e informações gerais, como a documentação e sua utilização pela comunidade de desenvolvimento. Além disso, foi realizado um comparativo entre as três arquiteturas analisados em todos os aspectos citados anteriormente e também entre o desempenho de cada uma por meio de testes de carga realizados em cima da API desenvolvida no estudo.

A ideia principal do estudo foi a realização do comparativo sem eleger uma arquitetura de desenvolvimento de API como sendo a melhor em relação as outras, mas sim buscando destacar a diferença de cada uma e para quais cenários de uso cada arquitetura encaixa melhor.

Os resultados apresentados pela solução implementada, demonstraram que dependendo do caso de uso executado pela API e/ou cenário de teste executado, cada arquitetura de desenvolvimento desempenhou ou não melhor que o outro, sem demonstrar um padrão de resultados onde somente uma arquitetura é superior às demais. Em testes relacionados a inserção de dados, GraphQL foi superior em relação às outras. Já a arquitetura gRPC ficou muito próxima a REST nos cenários de listagem, tendo inclusive um melhor desempenho no teste de listagem com menor quantidade de dados. A arquitetura REST obteve o melhor desempenho nos demais testes.

Como limitações do estudo, temos que a execução da solução e dos testes foram realizados exclusivamente em um ambiente local com uma rede estável e de baixa latência, o que não ocorre geralmente nos cenários tradicionais pois os projetos precisam estar hospedados em servidores que passam por instabilidades, dependem de navegadores de internet (browsers), aplicativos móveis, entre outros fatores que podem afetar o funcionamento e desempenho da aplicação. Além disso, a ferramenta de testes utilizada poderia ter sido mais explorada com mais cenários de testes e a API desenvolvida por mais que tenha sido desenvolvida com padrões de design de código, utilizou frameworks e bibliotecas que podem ou não ter influenciado nos resultados.

Por fim, apesar de REST dominar o cenário de desenvolvimento back-end até os dias atuais e ter obtido o melhor desempenho nos resultados do presente trabalho, este estudo traz como principal contribuição para o mercado de desenvolvimento a demonstração que há uma crescente busca de novas tecnologias, como GraphQL e gRPC, com o intuito de melhorar ou trazer novas opções que podem e já estão sendo aplicadas em diversos projetos.

Referências

BRASSCOM. Demanda de Talentos em TIC e Estratégia Σ TCEM. 2021. Disponível em: https://brasscom.org.br/pdfs/demanda-de-talentos-em-tic-e-estrategia-tcem/. Acesso em: 25 nov. 2023.

FIELDING, Roy Thomas. **Architectural Styles and the Design of Network-based Software Architectures**. 2000. 180 f. Dissertação (Doutorado) - Curso de Information And Computer Science, University Of California, Irvine, 2000.

GRPC. **GRPC**: a high performance, open source universal rpc framework get started!. 2024a. Disponível em: https://grpc.io/. Acesso em: 05 maio 2024.

GRPC. **About gRPC**: who is using grpc and why. 2024b. Disponível em: https://grpc.io/about/. Acesso em: 05 maio 2024.

GRPC. **Introduction to gRPC**: an introduction to grpc and protocol buffers. 2024c. Disponível em: https://grpc.io/docs/what-is-grpc/introduction/. Acesso em: 05 maio 2024.

GRPC. **Core concepts, architecture and lifecycle**: an introduction to key grpc concepts, with an overview of grpc architecture and rpc life cycle. 2024d. Disponível em: https://grpc.io/docs/what-is-grpc/core-concepts/. Acesso em: 05 maio 2024

GRAPHQL. **GraphQL**. 2024a. Disponível em: https://spec.graphql.org/draft/#sec-Overview. Acesso em: 10 abr. 2024.

GRAPHQL. Introduction to GraphQL. 2024b. Disponível em: https://graphql.org/learn/. Acesso em: 25 abr. 2024.

IBM. **Remote Procedure Call**. 2024. Disponível em: https://www.ibm.com/docs/en/aix/7.3?topic=concepts-remote-procedure-call. Acesso em: 05 maio 2024.

LANGUAGE Guide (proto 3): Covers how to use the version 3 of Protocol Buffers in your project. 2024. Disponível em: https://protobuf.dev/programming-guides/proto3/. Acesso em: 05 maio 2024.

MARTIN, Robert C.. **Arquitetura Limpa**: o guia do artesão para estrutura e design de software. Rio de Janeiro: Alta Books, 2019. 432 p. Traduzido por: Samantha Batista.

MDN WEB DOCS. **An overview of HTTP**. 2024a. Disponível em: https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview. Acesso em: 10 maio 2024.

MDN WEB DOCS. URL. 2024b. Disponível em: https://developer.mozilla.org/pt-BR/docs/Glossary/URL. Acesso em: 25 maio 2024.