Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Uma comparação de performance entre arquitetura GraphQL e REST

André Luiz de Moura Ramos Bittencourt

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Orientador
Prof. Dr. Vinícius Ruela Pereira Borges

Brasília
2021

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Uma comparação de performance entre arquitetura GraphQL e REST

André Luiz de Moura Ramos Bittencourt

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Prof. Dr. Vinícius Ruela Pereira Borges (Orientador)
CIC/UnB

Prof.a Dr.a Maristela Terto de Holanda
CIC/UnB

Prof.a Dr.a Vanessa Tavares Nunes
CIC/UnB

Prof. Dr. Marcelo Grandi Mandelli
Coordenador do Bacharelado em Ciência da Computação

Brasília, 25 de maio de 2021

Dedicatória

Dedico a todos que me apoiaram de alguma forma, seja com uma conversa, um ensinamento ou um conselho.

Agradecimentos

Agradeço inicialmente a minha família que sempre me apoiou em cada decisão que tomei, sem eles não teria chegado até esse momento. E em especial, agradeço o meu orientador Vinícius Borges por além de ter sido um grande mestre, foi também um grande ser humano e amigo no final da minha jornada na graduação.

Resumo

Cada vez mais o uso de plataformas web, aplicativos para celular e jogos online vem crescendo. Dados são transmitidos em um ritmo nunca antes visto e a internet ganha novos clientes a cada momento. Para ser capaz de lidar com tanto tráfego de rede, as aplicações precisam se comunicar de forma mais eficiente, em outras palavras, desenvolver melhores APIs.

Em 2000, Roy Fielding apresentou em seu doutorado o modelo *Representation State Transfer* (REST), que foi amplamente adotado pelo mercado e é o modelo mais seguido ainda nos dias de hoje. Entretanto, esse modelo se mostrou falho em alguns aspectos além de possuir pouca flexibilidade com relação à exposição de seus dados. Como forma de melhorar o modelo REST e resolver um problema interno da empresa, o Facebook desenvolveu uma nova arquitetura para APIs denominada GraphQL. Apesar de ser um modelo recente, a aprovação dessa nova tecnologia foi rápida e empresas como SpaceX, Airbnb e Twitter já a utilizam em seus projetos.

Nessa monografia, foi proposta uma metodologia composta por três métricas: tempo de resposta da requisição, utilização de CPU e consumo e memória. Os resultados mostraram que em situações onde se exige uma maior flexibilidade na busca pelos dados, a arquitetura GraphQL é uma escolha mais adequada, enquanto que em cenários em que os dados são utilizados sempre da mesma forma, a arquitetura REST é mais eficiente.

Palavras-chave: GraphQL, REST, API, Teste de performance

Abstract

The usage of Web platforms, mobile applications and online games are increasing every day. Data is transmitted faster than ever before, in every moment thousands of requests are going through the internet and to handle so much network traffic, applications need a more efficient communication with each other, in other words, better API's.

In 2000, Roy Fielding presented, in his doctoral thesis, the Representation State Transfer (REST). A model which was widely adopted by the market and today is one of the most used model. However, this model proves to be flawed in some aspects and it has little flexibility regarding the exposure of its data. In order to improve the REST model and solve an internal problem of the company, Facebook, developed a new architecture for API's which was called GraphQL. Despite being a recent model, the approval of this new technology was fast and companies like SpaceX, Airbnb and Twitter already use it in their projects.

In this monograph, the methodology was based in three metrics: request response time, CPU utilization and memory consumption. The results showed that, in scenarios where the system needs to fetch data in a more flexible way, GraphQL architecture is a more appropriate choice, while in scenarios where the data are always used in the same way, the REST architecture is more efficient.

Keywords: GraphQL, REST, API, Performance test

Sumário

1 Introdução **1**

1.1 Contexto 1

1.2 Objetivos 3

1.3 Estrutura do Documento 4

2 Fundamentação Teórica **5**

2.1 Application Programming Interface (API) 5

2.1.1 HyperText Transfer Protocol (HTTP) 5

2.1.2 Endpoint 6

2.1.3 Documentação 6

2.2 JavaScript Object Notation (JSON) 7

2.3 GraphQL 8

2.3.1 Linguagem de Consulta 9

2.3.2 Servidor 16

2.4 Representational State Transfer (REST) 19

2.5 Jmeter 22

2.6 Considerações Finais 22

3 Revisão de Literatura **23**

3.1 Estudos em Comparação de APIs 23

4 Metodologia Proposta **25**

4.1 Definição e Elaboração do Banco de Dados 26

4.2 API REST 27

4.3 API GraphQL 29

4.4 Configuração dos Testes 30

4.4.1 Teste com retornos iguais 30

4.4.2 Teste com retornos diferentes 31

4.4.3 Teste com múltiplas entidades 33

5 Resultados **35**
5.1 Especificação dos Testes 35
5.2 Teste com Retornos Iguais 36
5.2.1 Amostra de 100 funcionários 36
5.2.2 Amostra de 1000 funcionários 38
5.3 Teste com Retornos Diferentes 40
5.3.1 Amostra de 100 funcionários 40
5.3.2 Amostra de 1000 funcionários 43
5.4 Teste com Múltiplas Entidades 45
5.4.1 Amostra de 100 funcionários 46

6 Conclusão **49**
6.1 Considerações Finais 49
6.2 Trabalhos Futuros 50

Referências **51**

Lista de Figuras

1.1 Crescimento de pessoas usando a internet. 1

2.1 Servidor GraphQL. 9
2.2 Servidor REST. 20

4.1 Fluxograma detalhando as etapas da metodologia proposta. 25
4.2 Modelagem banco de dados. 26
4.3 Árvore do primeiro teste. 31
4.4 Árvore do segundo teste. 32
4.5 Árvore do terceiro teste. 33

5.1 Tempo de resposta para teste com retornos iguais em amostra de 100 funcionários. 36
5.2 Consumo de CPU para teste com retornos iguais em amostra de 100 funcionários. 37
5.3 Consumo de memória para teste com retornos iguais em amostra de 100 funcionários. 38
5.4 Tempo de resposta para teste com retornos iguais em amostra de 1000 funcionários. 38
5.5 Consumo de CPU para teste com retornos iguais em amostra de 1000 funcionários. 39
5.6 Consumo de memória para teste com retornos iguais em amostra de 1000 funcionários. 40
5.7 Tempo de resposta para teste com retornos diferentes em amostra de 100 funcionários. 41
5.8 Consumo de CPU para teste com retornos diferentes em amostra de 100 funcionários. 42
5.9 Consumo de memória para teste com retornos diferentes em amostra de 100 funcionários. 42

5.10 Tempo de resposta para teste com retornos diferentes em amostra de 1000 funcionários. 43
5.11 Consumo de CPU para teste com retornos diferentes em amostra de 1000 funcionários. 44
5.12 Consumo de memória para teste com retornos diferentes em amostra de 1000 funcionários. 45
5.13 Tempo de resposta para teste com múltiplas entidades em amostra de 100 funcionários. 46
5.14 Consumo de CPU para teste com múltiplas entidades em amostra de 100 funcionários. 47
5.15 Consumo de memória para teste com múltiplas entidades em amostra de 100 funcionários. 48

Capítulo 1

Introdução

1.1 Contexto

Nos últimos anos, foi presenciado um crescimento expressivo da quantidade de aplicações que consomem conteúdo via internet como aplicativos para aparelhos móveis, aplicações web, plataformas desktop, equipamentos conectados a rede, entre inúmeros outros sistemas e serviços que usam de forma direta e indireta dados trafegados pela internet. Esse grande crescimento é diretamente acompanhado pelo aumento de pessoas conectadas a rede [1] como mostra o Figura 1.1.

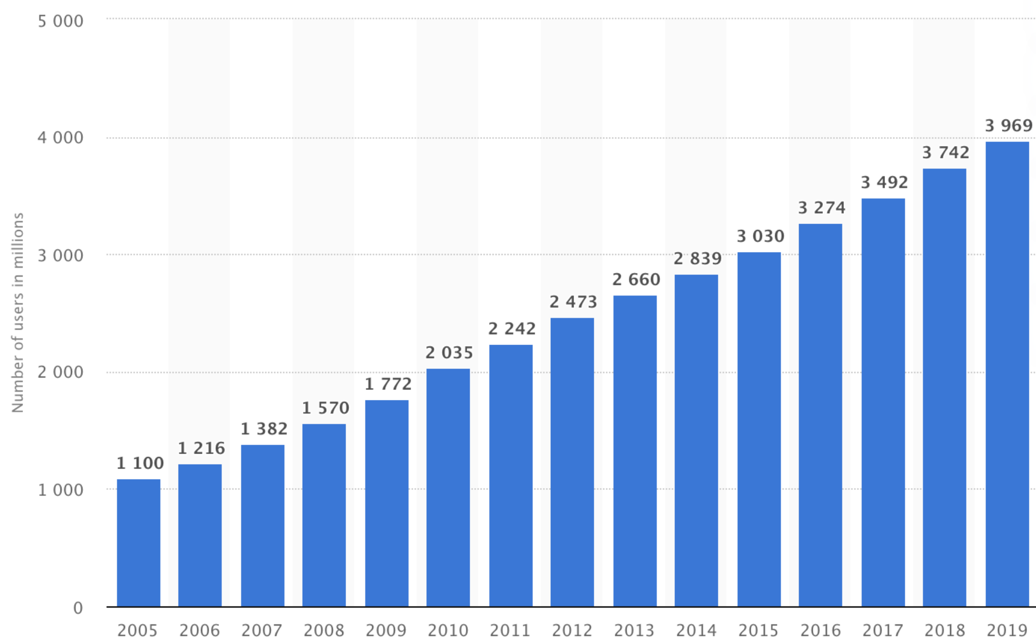

Figura 1.1: Crescimento de pessoas usando a internet.

O aumento da utilização da internet pela população demanda uma conexão mais rápida e para atender a esse desejo os sistemas de software se reinventam e buscam tecnologias e arquiteturas mais concisas e eficientes visando encontrar e melhorar pontos de ineficiência. Em um mundo onde cada vez mais a arquitetura de microsserviços [2] é usada e onde gradualmente mais dados são trafegados do *backend* para o *frontend* de uma aplicação, a comunicação entre serviços deve ser feita de forma eficiente para não se tornar o gargalo do sistema. A responsabilidade por fazer essa comunicação entre serviços é da *Application Programming Interface* (API).

Frequentemente, os serviços devem se comunicar para transmitir dados entre si, por exemplo, quando é feito a transferência de dinheiro de um banco para o outro. O sistema do primeiro deve ser capaz de se comunicar com o sistema do segundo para realizar a transferência bancária, ou quando é acessado uma página Web os dados que são vistos na tela são comumente obtidos de diferentes sistemas, unidos e mostrados pelo navegador. Para que essa comunicação seja feita de forma correta, os sistemas devem criar uma arquitetura que especifique informações, como os dados que serão transmitidos, o formato em que os dados vão ser trafegados, como será a transferência caso um erro ocorra, qual protocolo de rede será usado, entre inúmeras outras características. Uma vez determinado essa arquitetura, é necessário que as duas aplicações respeitem para que a comunicação seja feita de forma correta. Essa arquitetura criada nomeia-se de API. Ao longo do tempo, diferentes arquiteturas foram desenvolvidas com diferentes objetivos, seja aumentar a eficiência na comunicação, tornar mais simples o processo de desenvolvimento ou mesmo facilitar a transferência da informação.

No ano 2000, Roy Fielding em sua tese de doutorado [3], estava buscando padronizar a comunicação entre serviços independentemente de onde eles estivessem localizados. Em sua busca por uma padronização mundial, ele formulou a arquitetura *Representational State Transfer* (REST), que consiste em quatro princípios: contrato forte por meio da URL, não armazenamento de estado, modelo claro de cliente-servidor e capacidade de realizar cache. O modelo REST estabelece que deve ser documentado de forma clara o formato em que os dados vão ser transmitidos dado uma URL específica, essa padronização é importante para que o cliente tenha a certeza de que o que ele pedir vai chegar da forma como ele espera. O REST se baseia no conceito de não armazenamento de estado, significando que toda requisição é tratada de forma independente. Assim, uma requisição prévia não pode impactar uma requisição futura pois, no REST, elas não devem saber da existência uma da outra. Um modelo claro de cliente-servidor deve ser seguido, devendo haver limites claros entre as funções dos dois sistemas. Desta maneira, um servidor deve funcionar como produtor do dado e o cliente deve funcionar como consumidor. Uma característica resultante das duas primeiras mencionadas é o fato do modelo REST poder

realizar o cache de suas respostas. Sabendo-se que a URL segue um padrão de resposta e não há armazenamento de estado entre requisições, é possível salvar a resposta para uma requisição em memória e de forma rápida respondê-la para o cliente quando for solicitado. Devido a essa padronização mais clara fornecida pela arquitetura REST, empresas de tecnologia começaram a vender acesso a suas APIs ao mundo externo. Uma das primeiras a iniciar foi o EBay em novembro do ano 2000 [4], seguida anos depois pela Amazon em 2002 [5]. Com mais de 20 anos de existência, a arquitetura REST é o modelo mais popular no desenvolvimento de APIs.

Em 2012, o Facebook [6] começou o desenvolvimento dos seus aplicativos móveis de forma nativa, em que cada sistema operacional teria seu próprio código fonte. Até então, os aplicativos móveis usavam o mesmo código que era usado no aplicativo web, escritos em HTML. Embora no início essa reutilização de código funcionava bem, à medida que o aplicativo móvel foi ficando mais complexo, houve registros de baixo desempenho e problemas de funcionamento. Durante a transição do código para nativo, um grande problema surgiu: a API interna entrega o código direto em HTML, o que não era compatível com o desenvolvimento nativo. A primeira solução foi desenvolver uma API REST, que consumia dados de um banco relacional, mas inicialmente, foi entendido que essa arquitetura não era apropriada para o problema, pois a ideia consistia em trabalhar com dados em formato de grafos e não estruturados em tabelas e URLs. Essa barreira deu início ao projeto do GraphQL. De acordo com o Facebook o GraphQL foi a oportunidade de repensar a busca de dados de aplicativos móveis da perspectiva de designers e desenvolvedores de produtos [7]. O projeto foi considerado um sucesso e ainda no mesmo ano, sua primeira versão foi lançada junto com os aplicativos móveis em código nativo. Em 2015, o Facebook anunciou a primeira versão pública, para esse lançamento melhorias foram feitas, inconsistências corrigidas e uma especificação foi escrita para descrever o GraphQL e seu funcionamento. Em novembro de 2018, o projeto foi movido para a recém criada *GraphQL Foundation*, hospedada pela organização sem fins lucrativos *Linux Foundation* [8], atualmente tem seu código aberto disponível em repositório público.

1.2 Objetivos

Este trabalho tem como objetivo fazer uma comparação entre as principais características propostas pela arquitetura GraphQL perante a arquitetura REST. Para isso, duas APIs foram desenvolvidas da forma mais semelhante possível, uma vez que ambas consomem dados de um mesmo banco de dados, usam as mesmas tecnologias, apesar de cada uma seguir um padrão distinto de arquitetura.

Para realizar a comparação, foi levado em consideração três critérios, uso de CPU, consumo de memória e tempo de resposta. O objetivo é entender se, para esses critérios estabelecidos, as soluções oferecidas pela arquitetura GraphQL são mais eficientes que as soluções já existentes na arquitetura REST.

Com isso, foram feitos testes de performance para responder três perguntas:

- Qual arquitetura é mais performática para múltiplas requisições paralelas buscando exatamente os mesmos dados?
- A capacidade de diminuir a quantidade de campos em uma busca GraphQL o torna mais eficiente para a mesma busca em REST já que este não possui essa capacidade?
- O desempenho por meio de consultas alinhadas oferecidas pela arquitetura GraphQL é melhor do que o desempenho da arquitetura REST?

1.3 Estrutura do Documento

Este trabalho foi estruturado em cinco capítulos, o capítulo 2 apresenta a fundamentação teórica, explica-se os conceitos que dão base para o entendimento das arquiteturas de API que são usadas na comparação. O capítulo 3 realiza a revisão de literatura, explora-se artigos relacionados ao tema da monografia. O capítulo 4 desenvolve a metodologia, descreve-se os passos que foram feitos para a realização de cada teste. O capítulo 5 elabora os resultados, explica-se o ambiente, os cenários de teste e desenvolve-se gráficos com as métricas escolhidas. O capítulo 6 apresenta as conclusões e realiza sugestões para trabalhos futuros.

Capítulo 2

Fundamentação Teórica

Este capítulo busca introduzir terminologias, conceitos e tecnologias que fornecerão a base para o entendimento desse estudo. Será feito as definições de termos relevantes que abrangem o escopo de APIs, elaborado como as arquiteturas REST e GraphQL fazem requisições usam o protocolo HTTP com os dados sendo trefegados pela rede em formato JSON e, por fim, abordado as tecnologias que darão auxilio para a execução dos testes de performance.

2.1 Application Programming Interface (API)

O envio e recebimento de informações entre sistemas só é possível se o sistema que exporta essas informações possuir uma interface de comunicação bem definida e, para essa interface, é dado o nome de API. O acrônimo API vem do inglês *application programming interface*, e é uma abstração que todo sistema que deseja receber comunicação externa deve possuir. Para que essa comunicação ocorra bem é necessário que os sistemas tenham clareza de como chamar e receber requisições. Em outras palavras, é preciso que o sistema que irá exportar os dados por meio de sua API especifique como uma chamada deve ser feita e em qual formato os dados serão retornados. No contexto das arquiteturas REST e GraphQL, isso é feito utilizando o protocolo HTTP com a definição dos *endpoints* escritos na documentação.

2.1.1 HyperText Transfer Protocol (HTTP)

O *HyperText Transfer Protocol* (HTTP) existe a mais de 30 anos e atualmente é o principal meio de comunicação para sistemas Web [9]. O papel do HTTP, como descrito em seu nome, é determinar um protocolo para que a comunicação na rede entre cliente e servidor seja feita de forma adequada e padronizada. As duas arquiteturas que fazem parte deste

trabalho usam esse protocolo para trafegar seus dados e, por isso, a explicação de suas características básicas deve ser feita.

Estado

No protocolo HTTP não há armazenamento de estado entre duas requisições realizadas sucessivamente na mesma conexão. Isso significa que as requisições são independentes e que uma não sabe da existência da outra.

Métodos

Para definir o escopo dos propósitos de uma requisição, alguns métodos foram criados. Os mais populares são o GET e o POST. O método GET significa que a requisição deseja fazer apenas leitura de algum dado [10], enquanto o método POST pode encaminhar dados na requisição e executa uma escrita de informação [11]. Caso o método POST seja utilizado, seus dados são enviados por meio do campo *corpo* da requisição.

Cabeçalhos

Os cabeçalhos são campos de informações adicionais que uma requisição pode conter. Existem alguns cabeçalhos que tem seu nome padronizado, pois são comumente usados, como, por exemplo, o cabeçalho *Authorization*, que encaminha informações referentes a autenticação e autorização do cliente. A lista de cabeçalhos é extensa [12], mas ainda assim o protocolo permite a criação de cabeçalhos personalizados.

2.1.2 Endpoint

Endpoint, em português terminal, são os pontos de contato que uma API exporta para viabilizar outros sistemas a se comunicarem com ela. Esses pontos de contato podem ser representados por meio de URLs ou chamadas à funções remotas. Entretanto, independente da arquitetura, os *endpoints* estão na borda da API, recebendo as requisições.

2.1.3 Documentação

A documentação é uma das bases de uma API, como APIs em geral costumam exportar diferentes informações, fica inviável para quem for usa-lá entender qual o modelo de autenticação esperado, o que cada *endpoint* recebe e retorna, quais tipos de argumentos devem ser passados. Para isso, ao se desenvolver uma API, é necessário que se desenvolva um documento detalhando seu funcionamento.

2.2 JavaScript Object Notation (JSON)

O formato JSON é o padrão de transferência de dados usado pelas duas arquiteturas desse estudo. Ele é definido como uma coleção de pares chave-valor, conhecidos, normalmente, pelas linguagens de programação como dicionário, *hash table* ou objeto.

Um objeto começa com uma abertura de chaves (*{*) e termina com um fechamento de chaves (*}*). A chave é definida usando uma palavra, que pode conter espaços, seguido por dois pontos (*:*) para indicar o fim da chave e início do valor. Diferente da chave, o valor pode assumir um dos seguintes tipos de dados:

- Palavra
- Número
- Objeto (JSON)
- Vetor
- Booleano
- Null

Estes são tipos de dados universais. Virtualmente, todas as linguagens de programação modernas as suportam. Isso torna JSON um formato completamente independente de linguagem e, com isso, uma ótima estrutura para troca de dados [13].

O Exemplo 2.1 ilustra um objeto JSON utilizando cada um dos tipos de valores possíveis.

Exemplo 2.1: Objeto JSON.

```
{
    chave de texto: ‘‘um texto’’,
    chave_numerica: 123,
    objeto: {
        chave dentro do objeto: ‘‘estou aqui dentro’’,
        outra-chave-dentro: 12432
    },
    lista_de_numeros: [1, 2, 3, 4],
    verdade ou falso: true,
    ultima chave: null
}
```

2.3 GraphQL

GraphQL é uma linguagem de consulta para APIs que fornece uma descrição completa e compreensível dos dados, além de dar aos clientes o poder de buscar e modificar exatamente o que eles precisam em apenas uma requisição [14]. A arquitetura possui apenas um *endpoint* responsável por lidar com todas as requisições, normalmente usado como */graphql*. Esse *endpoint* aceita receber requisições HTTP empregando os métodos POST e, em alguns casos, o método GET.

Uma das grandes características do GraphQL é o fato do cliente poder escolher, no momento da requisição, os dados que ele gostaria de receber ou modificar. Para isso, a requisição deve conduzir as informações solicitadas pelo cliente até o servidor, que se encarregará de buscar e retornar os dados da maneira como foi pedida. Essa condução dos dados é feita de duas formas: no método GET, os dados são enviados usando uma consulta com parâmetros e no método POST, os dados são trafegados dentro o campo *corpo* da requisição [15]. Comumente, apenas o método POST é usado por conta da facilidade de transmitir os dados e ser mais apropriado com a proposta estabelecida pelo GraphQL [16]. Assim, este documento será pautado com foco nesse método.

Como mencionado, os dados são enviados no campo *corpo* da requisição, porém esse campo aceita transmitir informações em formatos variados [17]. O formato em que o dado será enviado é indicado pelo uso do cabeçalho *content-type* e no caso do GraphQL o valor contido nesse cabeçalho deve ser *application/json*, como especificado na documentação oficial [18].

Uma vez definido o *endpoint* GraphQL, o método POST e o cabeçalho *content-type* da requisição, basta escrever os dados que se deseja manipular ou modificar. Para isso, o GraphQL possui sua própria linguagem de consulta.

2.3.1 Linguagem de Consulta

A linguagem de consulta utilizada no GraphQL se assemelha a um objeto JSON que foi explicado na Seção 2.2 deste documento.

Suponha que exista uma API GraphQL, ilustrada pela Figura 2.1, que possui seu único *endpoint* com o caminho *"http://www.empresa.com.br/graphql"* e que retorna dados como nome, sobrenome, idade, email e telefone de todos os funcionários de uma empresa fictícia.

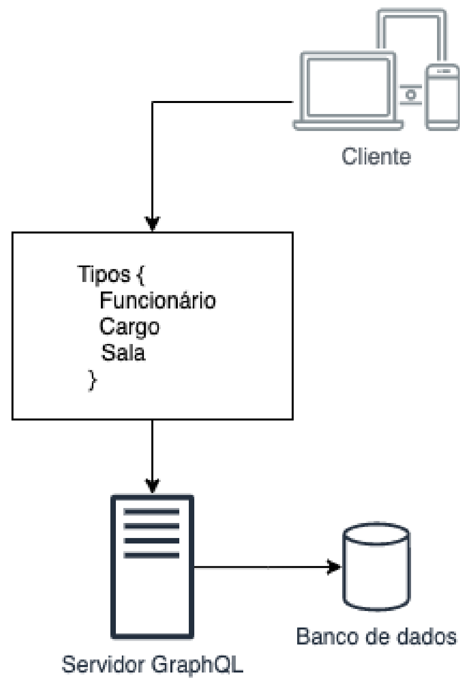

Figura 2.1: Servidor GraphQL.

O Exemplo 2.2 demonstra uma requisição que realiza uma consulta dos dados de um funcionário específico da empresa por meio dessa API.

Exemplo 2.2: Requisição de um funcionário em GraphQL.

```
URL = ``http://www.empresa.com.br/graphql''
metodo = POST
corpo = {
    query: ``{
        Funcionario(id: ``a1b'') {
            nome,
            sobrenome,
            idade,
            email,
            telefone,
            nascimento
        }
    }''
}
```

Toda requisição GraphQL deve possuir apenas uma chave chamada *query* e seu valor deve conter a informação solicitada no formato da linguagem definida por essa arquitetura. A primeira instrução definida pela linguagem GraphQL é que ela deve ser encapsulada por chaves, indicando que o primeiro caractere deve ser uma abertura de chaves e o último o fechamento de chaves. Logo em sequência, é necessário definir o *Tipo* do dado que se deseja buscar, em que no caso do exemplo, o *Tipo* é indicado pela palavra *Funcionario*. Um *Tipo* se assemelha a uma chamada de função em linguagens de programação, e dessa forma, pode ou não receber argumentos. Caso o *Tipo* aceite receber argumentos, o nome do argumento deve ser passado de forma explícita durante a chamada em um formato chave-valor, como mostrado no Exemplo 2.2 o argumento *id* é passado com o valor *a1b*. Por fim, deve-se definir os dados que se deseja buscar e para isso os *Campos* são usados, no exemplo o *Tipo Funcionario* possui os *Campos nome*, *sobrenome*, *idade*, *email*, *telefone* e *nascimento* [19].

A linguagem então é constituída por dois componentes principais, o *Tipo* que pode ou não possuir argumentos e o *Campo*.

Tipo

O *Tipo* é o modo como o GraphQL agrupa seus dados, sendo uma estrutura que, normalmente, possui um nome claro que indica de maneira simples o que ela representa. Um *Tipo* pode aceitar argumentos, o que o torna adaptável ao contexto do cliente, como no exemplo acima, em que o *Tipo Funcionario* possui o parâmetro *id* que determina o funcionário que se deseja buscar informações. Um *Tipo* pode possuir mais de um argumento e seus argumentos não necessariamente são obrigatórios.

A tipagem do argumentos é definida pelo servidor, e pode assumir as seguintes estruturas:

- Int: Inteiro com sinal de 32 bits;
- Float: Um valor de ponto flutuante de precisão dupla com sinal;
- String: Um sequência de caracteres codificados em UTF-8;
- Boolean: true ou false;
- ID: Representa um identificador exclusivo e serializado da mesma maneira que uma String.

O *Tipo* é a primeira estrutura que se define em uma requisição, pois é a partir dele que os dados que se deseja vão ser determinados. Além disso, uma requisição pode conter mais de um *Tipo*, em que deve-se alinhar o fim da chamada de um *Tipo* com o início do próximo. Por exemplo, suponha que nessa mesma API de exemplo exista o *Tipo sala*, responsável por retorna dados dos ambientes da empresa. Para buscar um funcionário e uma sala na mesma requisição, é preciso usar dois *Tipos* diferentes na mesma consulta, o Exemplo 2.3 demonstra essa requisição. O exemplo está pedindo dados do *funcionário* que possui o *id* igual a *"a1b"* e também os dados da sala que é identificada pelo numero *"483"*. Desta forma, é possível em apenas uma requisição buscar múltiplos *Tipos* diferentes e, consequentemente, múltiplos dados diferentes.

Exemplo 2.3: Requisição com dois Tipos distintos.

```
URL = ''http://www.empresa.com.br/graphql''
metodo = POST
corpo = {
    query: ''{
        Funcionario(id: ''a1b'') {
            nome,
            sobrenome,
            idade,
            email,
            telefone,
            nascimento
        }

        Sala(numero: 483) {
            andar,
            nome,
            tamanho
        }
    }''
}
```

Na linguagem GraphQL, caso se deseja chamar um *Tipo* sem passar argumentos, deve-se ocultar os parênteses. Assim, se na API fictícia houvesse um *Tipo Empresa* que não recebe parâmetros, sua chamada seria feita conforme o Exemplo 2.4.

Exemplo 2.4: Requisição sem parâmetros.

```
URL = ''http://www.empresa.com.br/graphql''
metodo = POST
corpo = {
    query: ''{
        Empresa {
            nome,
            endereco
        }
    }''
}
```

Além de *Campos*, os *Tipos* podem chamar outros *Tipos* em seu interior, o que torna a linguagem extremamente flexível. Para compreender esse conceito, suponha que na

API exista o *Tipo Cargo* que é responsável por retorna dados dos cargos de trabalho da empresa. O Exemplo 2.5 ilustra uma requisição que busca um cargo específico.

Exemplo 2.5: Requisição de um cargo específico.

```
URL = ``http://www.empresa.com.br/graphql''
metodo = POST
corpo = {
    query: ``{
        Cargo(nome: ``gerente'') {
            nivel,
            salario,
            habilidades
        }
    }''
}
```

Nesse caso, é recuperado os dados do *nível*, *salário* e *habilidades* do cargo identificado pelo nome gerente. Porém, todo funcionário dessa empresa fictícia possui um cargo atrelado a ele. Portanto, pode-se chamar o *Tipo Cargo* dentro do *Tipo Funcionario* e dessa forma tem-se em apenas uma requisição de forma concisa o *nome*, *sobrenome*, *salário* e *nível* do funcionário identificado pelo id *a1b*, como mostra o Exemplo 2.6.

Exemplo 2.6: Requisição de Tipos alinhados.

```
URL = ``http://www.empresa.com.br/graphql''
metodo = POST
corpo = {
    query: ``{
        Funcionario(id: ``a1b'') {
            nome,
            sobrenome,
            Cargo {
                salario,
                nivel
            }
        }
    }''
}
```

A inserção de um *Tipo* dentro de outro é feita de forma recursiva e não existe limites de camadas que a requisição pode assumir.

Mutation

Mutation é uma palavra reservada da linguagem para comunicar ao servidor que se deseja modificar dados. Seguindo o modelo do tópico anterior, suponha que se deseja fazer a modificação do nome de um funcionário, como ilustrado no Exemplo 2.7.

Exemplo 2.7: Requisição para modificar funcionário.

```
URL = ``http://www.empresa.com.br/graphql''
metodo = POST
corpo = {
  query:
    ``mutation {
      ModificarFuncionario(nome: ``Andre'') {
        status
      }
    }''
}
```

Como já mostrado no tópico anterior, deve-se passar um JSON com apenas um campo chamado *query*. Porém, ao invés de chamar diretamente os *Tipos* como era feito, deve-se primeiro usar a palavra reservada *mutation* e, entre chaves, escrever os *Tipos* que se deseja modificar. A sintaxe usada é a mesma já estabelecida, mas que, nesse caso os argumentos que os *Tipos* vão receber serão condizentes com os dados que se deseja modificar. No Exemplo 2.7 o *Tipo ModificarFuncionario* pode aceitar outros argumentos, mas apenas o nome está sendo requisitado para modificação. Além de passar os argumentos, é necessário definir os *Campos* que se deseja receber como resposta da requisição, em que no Exemplo 2.7, apenas o *Campo status* será retornado após a requisição ser concluída.

Campos

O *Campo* é a maneira que o cliente define os dados que deseja buscar dentro de um *Tipo*. Cada *Campo* possui sua própria tipagem e quem a define é a API, a tipagem varia entre os seguintes elementos:

- Int: Inteiro com sinal de 32 bits;
- Float: Um valor de ponto flutuante de precisão dupla com sinal;

- String: Um sequência de caracteres codificados em UTF-8;
- Boolean: true ou false;
- ID: Representa um identificador exclusivo e serializado da mesma maneira que uma String.

Resposta

Uma vez entendido os conceitos de como realizar uma requisição GraphQL utilizando sua linguagem, deve-se entender como os dados solicitados são encaminhados por meio da resposta. Da mesma maneira que a requisição é encaminhada seguindo um padrão do formato JSON, a resposta também segue um padrão e também é codificada em JSON. Para entender melhor o funcionamento de como a resposta é retornada, tenha como base o Exemplo 2.8.

Exemplo 2.8: Requisição de um funcionário específico.

```
URL = ``http://www.empresa.com.br/graphql''
metodo = POST
corpo = {
    query: ``{
        Funcionario(id: ``a1b'') {
            nome,
            sobrenome,
            idade,
            email,
            telefone,
            nascimento
        }
    }''
}
```

Após o servidor processar a requisição, a resposta retornada é ilustrada no Exemplo 2.9.

Exemplo 2.9: Resposta de uma requisição GraphQL.

```
{
    data: {
        Funcionario: {
            nome: ''Andre'',
            sobrenome: ''Bittencourt'',
            idade: 22,
            email: ''emailAndre@mail.com'',
            telefone: ''34233423'',
            nascimento: ''2000-11-23''
        }
    }
}
```

Toda resposta GraphQL possui a chave *data*, que é responsável por retornar os dados que foram solicitados. As chaves contidas no campo *data* tem o mesmo nome dos *Tipos* que foram requisitados, como no exemplo acima, o retorno possui a chave *Funcionario* pois ele foi especificado na requisição. Os *Tipos* buscados são retornados em formato de objeto e as chaves contidas nesse objeto possuem os mesmos nomes dos campos solicitados na requisição. No Exemplo 2.9 foi requisitado o nome do funcionário no primeiro campo, logo o primeiro campo da resposta possui *nome: "Andre"*.

2.3.2 Servidor

Essa seção é responsável por esclarecer o papel do servidor GraphQL. Será abordado como é feito a criação de *Tipos* e *Campos* e também quais passos são dados desde do recebimento de uma requisição até sua resposta.

Schema

Schema, em português esquema, é onde todos os *Tipos* são definidos para serem expostos pelo servidor GraphQL. Diferentemente do cliente, que escolhe quais *Campos* passar, no *Schema*, todos os *Campos* devem ser implementados para justamente poder lidar com qualquer modelo de requisição feita. É nele que é estabelecido a tipagem de cada *Campo* e também a relação entre os *Tipos* e, assim como existe a linguagem de consulta GraphQL,

também existe uma linguagem própria de escrita e, não por acaso, elas apresentam semelhanças.

O *Schema* possui dois grandes *Tipos* que são próprios da linguagem e que toda API GraphQL deve implementar: o *Tipo Query* e o *Tipo Mutation*. Todos os outros *Tipos* implementados serão subtipos desses dois. Os *Tipos* que estiverem abaixo do *Tipo Query* poderão apenas consultar dados na API e por outro lado, os *Tipos* pertencentes ao *Mutation* causarão um efeito de mudança. Para entender melhor como ela funciona, um exemplo é colocado. Seguindo com o modelo de API proposto no início da seção, a definição do *Tipo Funcionario* e *Cargo* são feitas da seguinte maneira:

```
type Funcionario {
    id: ID!
    nome: String!
    sobrenome: String
    idade: Int
    email: String!
    telefone: String
    nascimento: Date
    cargo: Cargo!
}

type Cargo {
    id: ID!
    nome: String!
    salario: Float!
    nivel: Int
}
```

A definição de todos os *Tipos* começa com a palavra *type* seguida do nome do *Tipo*. Em sequência, é definido os *Campos* que o *Tipo* possui em um formato de chave-valor, em que a chave é o nome do *Campo* e o valor a tipagem do *Campo*. Ao se escrever um ponto de exclamação após a tipagem do campo, é definido que aquele *Campo*, nunca retornará um valor nulo. Dentro do *Tipo Funcionario* é definido o *Campo cargo* que possui como tipagem *Cargo*. Essa tipagem é uma referência para o *Tipo Cargo* definido nesse mesmo Schema. Na linguagem GraphQL, a interconexão entre os *Tipos* é feita usando seus nomes como referência. Uma vez definidos os *Tipos*, é necessário estabelecer quais argumentos cada *Tipo* recebe e, para isso, deve-se encapsular cada *Tipo* dentro dos *Tipos* macros

definidos pela linguagem. Como o *Tipo Funcionario* e *Cargo* fazem apenas consulta e podem receber um identificador como argumento, a definição final da linguagem fica da seguinte forma:

```
type Funcionario {
    id: ID!
    nome: String!
    sobrenome: String
    idade: Int
    email: String!
    telefone: String
    nascimento: Date
    cargo: Cargo!
}

type Cargo {
    id: ID!
    nome: String!
    salario: Float!
    nivel: Int
}

type Query {
  Funcionario(id: ID!): Funcionario
  Cargo(nome: String!): Cargo
}
```

Dentro do *Tipo Query*, é estabelecido em formato de chave-valor, em que na chave é identificado o nome do *Tipo* que o cliente deve usar no momento da requisição, assim como quais argumentos podem ser passados, e no valor é identificado qual *Tipo* será retornado.

Resolver

Uma vez definido o *Tipo Query* e o *Tipo Mutation*, é preciso realizar a lógica responsável por buscar e agrupar esses dados da forma como foi pedida. O *Resolver* é a estrutura responsável por receber as requisições no servidor, elaborar a lógica e retornar o dado da maneira correta para o cliente que a requisitou. O *Resolver* usa como base as estruturas definidas no *Schema*, mais especificamente cada *Tipo* dentro de *Query* e *Mutation* deverá

ter uma definição de forma idêntica na estrutura do *Resolver*. Isso significa que se no *Schema* o *Tipo Funcionario* recebe o argumento *id*, o *Resolver* deve possuir uma função que lida com essa requisição e que deve receber o mesmo argumento.

A função que lida com um *Tipo* pode fazer isso na maneira que achar mais adequado para o contexto, o que torna o GraphQL muito flexível. Por exemplo, o *Resolver* do *Tipo Funcionario* pode buscar os dados que precisa em um banco de dados relacional enquanto o *Resolver* do *Tipo Cargo* pode requisitar outra API que irá prover a resposta. Cada linguagem de programação possui sua própria maneira de arquitetar a lógica da implementação do *Resolver*. A única exigência é que toda API GraphQL deve implementar um *Resolver* para cada *Tipo* definido no *Schema*.

2.4 Representational State Transfer (REST)

Representational State Transfer (REST), em tradução livre, Transferência de estado representacional, define um conjunto de boas práticas utilizando requisições HTTP para tornar fácil a compreensão e desenvolvimento de diferentes API's.

O modelo REST realiza a organização dos *endpoints* por meio de diferentes URL's, usufruindo dos diferentes métodos http, ou seja, dentro da arquitetura REST cada método http tem sua função e escopo definido. Os quatro principais métodos usando são:

- **GET**: esse método é usado quando se deseja apenas buscar informações, não sendo possível passar informações por meio do campo *corpo* da requisição [20]. Quando se deseja passar parâmetros, é usado ou o modelo de consulta por parâmetros, ou diretamente por meio da URL seguindo o padrão definido por quem desenvolveu a API.

- **POST**: esse método é usado quando se deseja adicionar uma nova informação. Os dados são passados usando o *corpo* da requisição no formato JSON e deve seguir o padrão definido por quem desenvolveu a API para que a adição dos dados aconteça de forma correta.

- **PUT**: esse método é usado quando se deseja modificar uma informação. Os dados que se deseja atualizar são enviados, em formato JSON, no *corpo* da requisição e deve seguir o padrão definido pela API em consideração para que a modificação dos dados aconteça de forma correta.

- **DELETE**: esse método é usado quando se deseja deletar uma informação. A passagem de informação por meio do *corpo* da requisição é , apesar de não ser recomendável. Comumente, a deleção de uma informação possui uma URL, em que é possível escolher os parâmetros que serão passados.

A documentação de uma API REST é de vital importância para que outras aplicações possam utilizá-la. A documentação é única para cada API e deve explicar como é o funcionamento de cada *endpoint*, ou seja, informações como a URL que deve ser usada, o formato dos parâmetros que podem ser passados, o método que deve ser aplicado, se é necessário autenticação, dados e o formato da resposta.

Suponha que exista uma API REST, ilustrada pela Figura 2.2, que nos retorne dados como nome, sobrenome, idade, email e telefone de todos os funcionários de uma empresa fictícia.

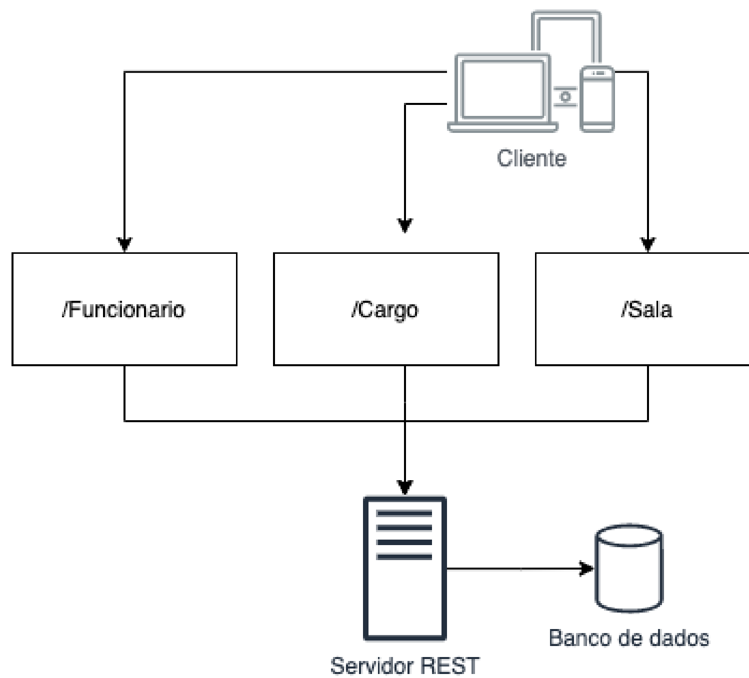

Figura 2.2: Servidor REST.

O Exemplo 2.10 realiza uma consulta dos dados de um funcionário específico da empresa por meio dessa API. Por ser uma consulta, é empregado o método GET e passado a identificação do funcionário por meio da URL.

Exemplo 2.10: Requisição REST para buscar funcionário.

```
URL = ``http://www.empresa.com.br/api/funcionarios/a1b''
metodo = GET
```

O primeiro ponto a se destacar no Exemplo 2.10 é que todos os *endpoints* em uma API REST possuem uma URL base. Nesse caso, a URL base é *"http://www.empresa.com.br/api"*, significando que todos os *endpoints* serão iniciados com esse padrão. O segundo ponto é que a identificação do funcionário é feita por meio da URL. No caso do Exemplo 2.10, a identificação aparece após o caminho *funcionarios*. O Exemplo 2.11 demonstra o retorno após executar essa requisição.

Exemplo 2.11: Resposta requisição REST.

```
{
    nome: ''Andre'',
    sobrenome: ''Bittencourt'',
    idade: 22,
    email: ''emailAndre@mail.com'',
    telefone: ''34233423''
}
```

Cada API indicará o formato do retorno dos dados em sua documentação. No caso desta API fictícia, os dados são retornados no formato JSON com os campos explícitos na resposta. Uma API REST não possui padronização com relação ao retorno da informação, sendo assim, para a composição dos testes que serão feitos neste documento, é admitido retornos apenas no formato JSON.

Assim como qualquer sistema, uma API também está suscetível a erros e no caso de API REST o tipo do erro é indicado por meio do atributo *Status Code*. Portanto, além do retorno da informação, esse atributo também é encaminhado na resposta. O *Status Code* é um número que varia de 100 a 599, em que cada um possui um significado específico do que ocorreu na requisição. O mais comum é o *Status Code* igual a 200 indicando que a requisição foi bem sucedida, enquanto que o *Status Code* igual a 500 indica um erro no servidor e 404 quando algo não foi encontrado [21].

Para exemplificar o método POST, é adicionado um novo funcionário. Suponha que para adicionar um funcionário a documentação indica que deve-se usar a URL "http://www.empresa.com.br/api/funcionarios/" e que os campos devem ser passados no corpo da requisição, demostrado no Exemplo 2.12.

Exemplo 2.12: Adição de funcionário API REST.

```
url = ''http://www.empresa.com.br/api/funcionarios/''
metodo = POST
corpo = {
    nome: ''Ricardo'',
    sobrenome: ''Silva'',
    idade: 30,
    email: ''emailRicardo@mail.com'',
    telefone: ''89329823''
}
```

Após a requisição, uma resposta é obtida com *Status Code* 201, indicando que ocorreu uma criação na API.

2.5 Jmeter

Jmeter é uma ferramenta *open source* responsável por realizar diferentes tipos de teste em um sistema de *software*, sendo os mais populares: testes de performance, carga e stress.

Para iniciar a configuração de um teste no Jmeter, deve-se inicialmente criar um *Plano de Teste* que, para esse trabalho, é denominado de "árvore de teste". Essa árvore é responsável por organizar o fluxo do teste por meio de elementos, os principais elementos são:

- *Thread Elements*: Responsável por definir características do teste como quantidade de usuários simultâneos, a quantidade de requisições por usuário entre outras propriedades usualmente controladas em testes de performance.
- *Config Elements*: Responsáveis por realizar a requisição HTTP. Nesse grupo, encontram-se elementos para ajustar cabeçalhos, modificar *cookies*, definir a URL do servidor.
- *Logic Controllers*: Elementos responsáveis por lidar com controle de fluxo como laços de repetições e verificação de valores.
- *Listeners*: Elementos de resposta da requisição, usualmente todos os testes os possuem visto que são os responsáveis por mostrar os resultados. Cada elemento possui suas características próprias de visualização com diferentes parâmetros podendo ser coletados.

2.6 Considerações Finais

A arquitetura REST possui um modelo mais simples tanto para o lado do cliente quanto para o lado do servidor comparado a arquitetura GraphQL. Um dos objetivos desse trabalho é verificar se essa complexidade promovida pela arquitetura GraphQL possui vantagens na performance e, para isso, o Jmeter irá auxiliar com o desenvolvimento dos testes por ser capaz de configurar, com precisão, cada requisição ao servidor.

No capítulo seguinte será elaborado as etapas tomadas para preparar o sistema para a execução dos testes.

Capítulo 3

Revisão de Literatura

Neste capítulo, foi feito o levantamento de estudos relacionados por meio da leitura de diferentes artigos. Procura-se compreender os fatores importantes para uma comparação entre APIs, quais ferramentas usar para realizar testes de performance e as métricas mais relevantes para serem coletadas.

3.1 Estudos em Comparação de APIs

Devido ao GraphQL ser um modelo recente no mercado, ainda existem poucas pesquisas na literatura e grande parte não abordou as maiores vantagens oferecidas por essa nova arquitetura. Thomas Eizinger [22] faz uma comparação ampla entre as duas arquiteturas em questão. Neste artigo, além de abordar critérios de performance, também é abordado critérios a nível de código como simplicidade de codificação, capacidade de reutilização de código, modularização e responsabilidade dos componentes. Apesar do objetivo do trabalho do autor não ser exclusivamente em teste de performance, o artigo foi importante para entender a maneira como a comparação foi realizada. O trabalho realizado pelo autor ficou bastante completo, mas não foi abordado a vantagens que o modelo GraphQL oferece nos testes de performance.

Camille Oggier [23] conduziu uma comparação mais direta de performance entre as arquiteturas. O objetivo do artigo é verificar se a arquitetura GraphQL é mais otimizada do que a arquitetura REST. Para realizar a análise, a autora emprega a métrica de tempo de resposta das requisições e tamanho, em bytes, do corpo da requisição. As chamadas às APIs são feitas por meio de um cliente programado em React [24], o que busca simular um ambiente real de cliente servidor. Apesar da simulação de um ambiente Web trazer vantagens, ele também limita um controle melhor do teste utilizando ferramentas próprias como, *Jmeter* ou *Gatling*, em que é possível calcular com maior precisão as métricas utilizadas. O trabalho realizou apenas dois cenários de testes, em que o primeiro considera

duas arquiteturas que buscam os mesmos dados em apenas uma requisição, enquanto que o segundo aborda a vantagem de alinhamento de consulta proposta pelo GraphQL. Em geral, as conclusões propostas pela autora são similares com as conclusões estabelecidas nessa monografia.

Por fim, Mafalda Landeiro [25] realiza uma comparação mais profunda em termos de performance entre as duas arquiteturas. É levado em consideração cenários mais amplos, como o problema de *Over Fetching* e *Under Fetching*, que a arquitetura REST possui. Também é abordado as vantagens que a arquitetura GraphQL oferece como o alinhamento de consulta e a seleção dos campos. Os critérios de avaliação levantados no artigo foram: tempo e tamanho da resposta das requisições. Assim como nessa monografia, a autora usou ferramentas próprias para testes de performance, no caso do artigo foi utilizado a ferramenta *Jmeter*. O foco do artigo consistiu em investigar se o modelo GraphQL é capaz de melhorar a performance e ao mesmo tempo oferecer uma flexibilidade na busca dos dados. A conclusão do artigo para esse questionamento comprova esse fato, pois esse novo modelo é capaz de trazer performance e flexibilidade. Como esse artigo apresentou uma metodologia e experimentos completos em termos de comparação de performance, ele foi adotado como base para o desenvolvimento dessa monografia.

Capítulo 4

Metodologia Proposta

Este capítulo descreve a metologia que será abordada na comparação entre as duas APIs, é descrito como cada API foi estruturada, a base de dados que elas estão utilizando e a forma que o Jmeter foi configurado para realizar o teste de performance. Como o objetivo é realizar uma comparação justa, as duas APIs foram desenvolvidas usando as mesmas tecnologias, ambas escritas em Javascript, sendo interpretadas e executadas pelo *runtime* Node.js, usando o *framework* Express e retornando dados em formato JSON.

Para a elaboração da pesquisa, dividiu-se a metodologia em cinco etapas representadas na Figura 4.1. As etapas, com exceção da última, serão explicadas individualmente em cada seção desse capítulo. A etapa final, que descreve os resultados e conclusões, será explicado no próximo capítulo.

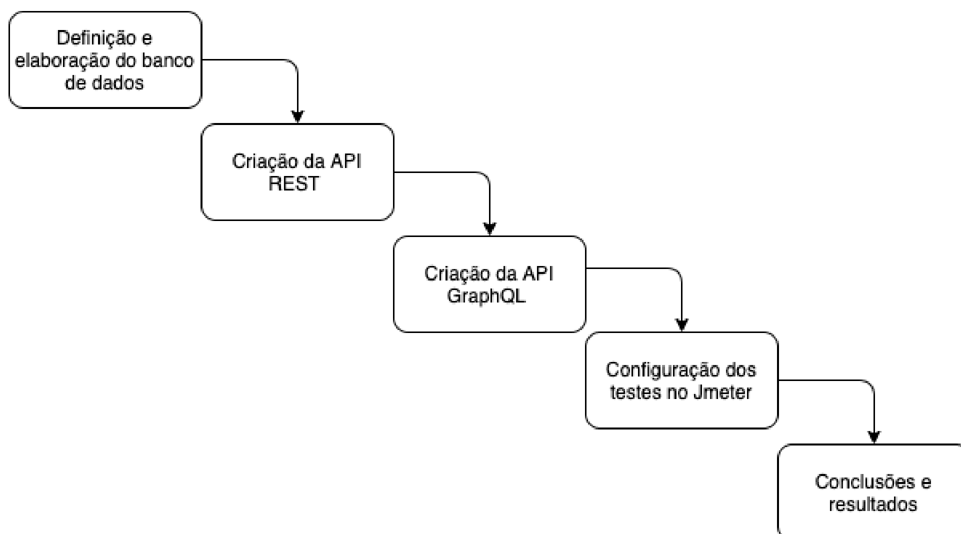

Figura 4.1: Fluxograma detalhando as etapas da metodologia proposta.

4.1 Definição e Elaboração do Banco de Dados

O banco de dados possui um papel importante no teste, uma vez que como as duas APIs irão usá-lo, ele deve ser construído de forma a não beneficiar nenhuma das arquiteturas. Para isso, foi elaborado um banco de dados que possui três entidades com o objetivo de simular informações de uma empresa fictícia. As entidades possuem variados campos e, entre eles, campos que possuem referência a outras entidades fazendo com que as entidades se tornem dependentes umas das outras.

A construção foi feita usando o banco de dados não relacional MongoDB e sua estrutura é mostrada na Figura 4.2.

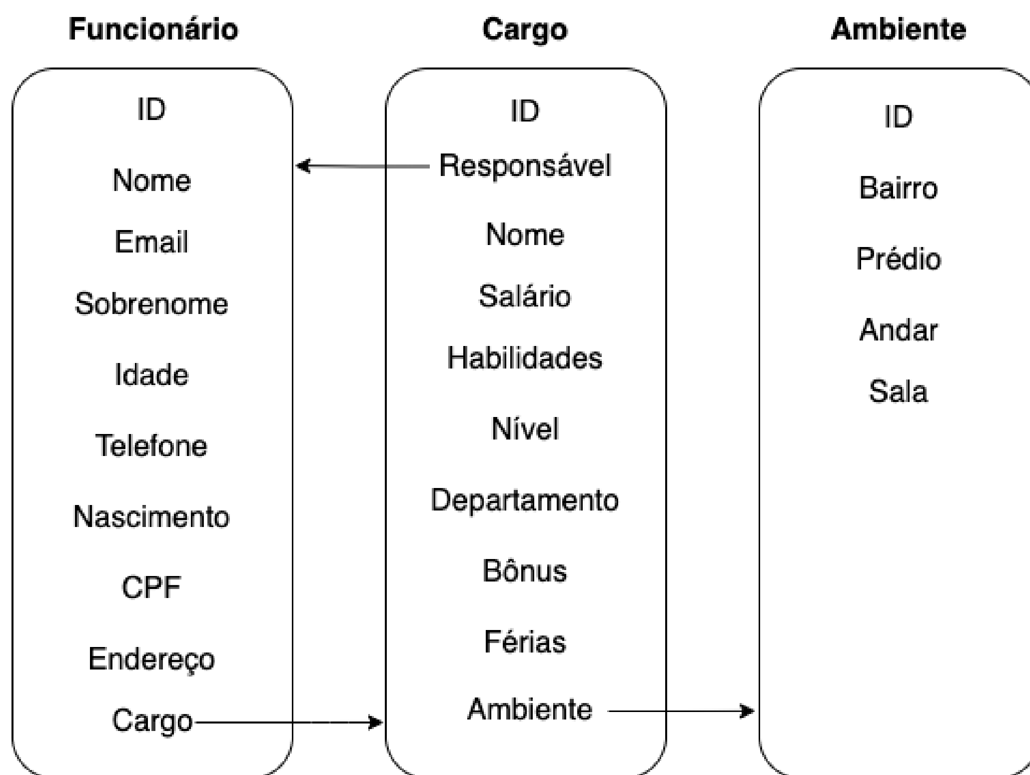

Figura 4.2: Modelagem banco de dados.

A primeira *collection* chamada **Funcionário** possui os seguintes campos:

- **ID**: Identificador único do funcionário
- **Nome**: Nome do funcionário
- **Email**: Email do funcionário
- **Sobrenome**: Sobrenome do funcionário
- **Idade**: Idade do funcionário

- **Telefone**: Telefone do funcionário
- **Nascimento**: Data de nascimento do funcionário
- **CPF**: CPF do funcionário
- **Endereço**: Endereço residente do funcionário
- **Cargo**: ID do **Cargo** que o funcionário possui

A segunda *collection* chamada **Cargo** possui os seguintes campos:

- **ID**: Identificador único do cargo
- **Responsável**: ID do **Funcionário** responsável por gerenciar o cargo
- **Nome**: Nome do cargo
- **Salário**: Salário médio do cargo
- **Habilidades**: Habilidades separadas por vírgulas necessárias para assumir o cargo
- **Nível**: Nível necessário para assumir o cargo
- **Departamento**: Departamento de atuação do cargo
- **Bônus**: Identifica se o cargo possui sistema de bonificação ou não
- **Férias**: Quantidade de dias de férias que o cargo possui
- **Ambiente**: ID do **Ambiente** em que os funcionários do cargo trabalham

A terceira *collection* chamada **Ambiente** possui os seguintes campos:

- **ID**: Identificador único do ambiente
- **Bairro**: Bairro em que o ambiente está localizado
- **Prédio**: Prédio em que o ambiente está localizado
- **Andar**: Andar em que o ambiente está localizado
- **Sala**: Sala em que o ambiente está localizado

4.2 API REST

Para a elaboração deste projeto, foi desenvolvida uma API REST com três *endpoints* que consomem dados provindos do banco de dados especificado na seção anterior. Como

mencionado no início do capítulo, o código da API foi escrito em Javascript rodando em Node.js e usando Express.

Todos os *endpoints* possuem a mesma URL base e a descrição do caminho será feita relativo a essa base. As características de cada *endpoint* são descritas a seguir:

- O primeiro *endpoint* tem como objetivo buscar e retornar todos os funcionários da empresa:
  - **URL**: /funcionarios
  - **Método**: GET
  - **Retorno**: Esse *endpoint* retorna um vetor de funcionário, onde cada elemento do vetor é um objeto com os campos: ID, Nome, Email, Sobrenome, Idade, Telefone, Nascimento, CPF, Endereço e Cargo.

    O campo ID é um identificador único do funcionário e o campo Cargo é um identificador único do cargo desse funcionário.
- O segundo *endpoint* tem como objetivo buscar e retornar informações de um cargo específico da empresa:
  - **URL**: /cargo/<ID> onde <ID> deve ser o ID do cargo que se deseja buscar
  - **Método**: GET
  - **Retorno**: Esse *endpoint* retorna um objeto com os campos: ID, Responsável, Nome, Salário, Habilidades, Nível, Departamento, Bônus, Férias e Ambiente.

    O campo ID é um identificador único do cargo, o campo Responsável é um identificador único de um funcionário e o campo Ambiente é um identificador único de um ambiente de trabalho.
- O terceiro *endpoint* tem como objetivo buscar e retornar informações de um ambiente específico da empresa:
  - **URL**: /ambiente/<ID> onde <ID> deve ser o ID do ambiente que se deseja buscar
  - **Método**: GET
  - **Retorno**: Esse *endpoint* retorna um objeto com os campos: ID, Bairro, Prédio, Andar e Sala.

    O campo ID é um identificador único do Ambiente.

4.3 API GraphQL

A API GraphQL que foi desenvolvida para esse projeto possui três Schemas, três *Tipos* e três Resolvers que consomem dados provindos do banco de dados especificado na seção anterior. Como mencionado no início do capítulo, o código da API foi escrito em Javascript rodando em Node.js, usando Express. Diferentemente da arquitetura REST, o GraphQL necessita do auxílio de uma biblioteca adicional para lidar com requisições, para isso foi usado a biblioteca Apollo. Os *Schemas* definidos são:

```
type Funcionario {
    id: ID
    nome: String
    email: String
    sobrenome: String
    idade: Int
    telefone: String
    nascimento: String
    CPF: String
    endereço: String
    cargo: Cargo
}

type Cargo {
    id: ID
    responsavel: Funcionario
    nome: String
    salario: Float
    habilidades: String
    nivel: Int
    departamento: String
    bonus: Float
    ferias: Int
    ambiente: Ambiente
}

type Ambiente {
    id: ID
    bairro: String
```

```
    predio: String
    andar: Float
    sala: String
}
```

Para a elaboração deste experimento, a API GraphQL possui apenas *Tipos* de busca, definidos como:

- **Funcionários** que não recebe argumentos e retorna um vetor de funcionários;
- **Cargo** que recebe o campo ID como argumento e retorna um cargo;
- **Ambiente** que recebe o campo ID como argumento e retorna um ambiente.

4.4 Configuração dos Testes

O trabalho se propõe a responder três perguntas diferentes, em que cada pergunta possui três cenários de teste independentes. Será descrito cada cenário em detalhes nessa subseção.

4.4.1 Teste com retornos iguais

O primeiro cenário de teste consiste em verificar e comparar o comportamento das APIs quando os mesmos dados são requisitados.

Para criar um cenário justo, são feitas requisições de forma que as duas APIs façam a mesma quantidade de consulta ao banco de dados. O objetivo desse teste é entender o comportamento das estruturas, e como o tempo de processamento das requisições ao banco são custosas, se uma API fizer mais requisições do que outra isso enviesará o resultado.

Com essas premissa definida, o *endpoint* REST escolhido foi o */funcionarios* que retorna os campos ID, Nome, Email, Sobrenome, Idade, Telefone, Nascimento, CPF, Endereço e Cargo. Uma vez definido o *endpoint* REST, é configurado a chamada GraphQL utilizando o *Tipo Funcionario* e selecionando os campos corretos para serem condizentes com os campos retornados pela API REST.

O Jmeter é configurado para elaborar as requisições e a Figura 4.3 ilustra a árvore de teste modelada.

Figura 4.3: Árvore do primeiro teste.

Foi feito o desenho de cada API em *Threads* diferentes, pois é possível desabilitar uma *Thread* e executar o teste em apenas uma das APIs de forma isolada. Os campos das duas *Threads* são semelhantes, e apesar de internamente possuir configurações diferentes, seus objetivos são iguais.

O campo **Variáveis** foi inserido para organizar o teste, sendo o responsável por definir atributos que vão ser usados em cada API como a URL do servidor e a porta que ele está usando. O campo **Cabeçalhos HTTP** indica os cabeçalhos que cada API irá enviar nas requisições. Nesse caso, as duas APIs enviam o mesmo cabeçalho *"Content-Type: application/json"*. Isso é necessário para que a requisição funcione de forma correta. O campo **Funcionarios** representa a própria requisição, em que esse campo armazena as variáveis e cabeçalhos definidos previamente. No caso, a API REST é requisitado apenas o *endpoint /funcionarios* usando o método GET. Já na requisição GraphQL, a *query* para buscar o *Tipo* funcionário foi elaborada de modo a buscar os mesmos campos retornados pela API REST. O campo **Resultado** é responsável por capturar as informações de resposta das requisições. Nesse campo, é usando o *Result in Table* que armazena informações como tempo de resposta, quantidade de bytes enviado, quantidade de bytes recebido, latência e informações adicionais que não serão abordadas nessa pesquisa.

4.4.2 Teste com retornos diferentes

O segundo cenário de teste consiste em verificar e comparar o comportamento das APIs quando a API GraphQL requisita menos dados que a API REST. O objetivo desse teste é usar da habilidade de selecionar os campos que o GraphQL possui para entender se existe uma melhora em sua performance, uma vez que agora ele irá requisitar menos dados que a API REST.

Tomando a mesma precaução que foi feito no primeiro teste, o objetivo é ter um cenário justo e, com isso, é importante garantir que as duas APIs façam a mesma quantidade de requisições ao banco de dados já que essas requisições são custosas e podem afetar o comportamento do resultado. Para esse teste é usado o *endpoint /funcionarios* da API REST como referência, ele retorna os campos ID, Nome, Email, Sobrenome, Idade, Telefone, Nascimento, CPF, Endereço e Cargo. Com esse *endpoint* definido, configura-se a chamada GraphQL usando o *Tipo Funcionario*, além de selecionar uma quantidade menor de campos do que o retornado pela API REST.

O Jmeter é configurado para elaborar as requisições e a Figura 4.4 ilustra a árvore de teste modelada. Seguindo o padrão proposto pelo primeiro teste, é desenhado cada API em *Threads* diferentes para facilitar os testes em apenas uma das APIs de forma isolada.

Figura 4.4: Árvore do segundo teste.

O campo **Variáveis** é responsável por definir atributos que serão usados em cada API como a URL do servidor e a porta que ele está usando. O campo **Cabeçalhos HTTP** indica os cabeçalhos que cada API irá enviar nas requisições, nesse caso as duas APIs enviam o mesmo cabeçalho *"Content-Type: application/json"*. O campo **Funcionarios** representa a própria requisição. Na API REST é requisitado apenas o *endpoint /funcionarios* usando o método GET, já na requisição GraphQL a consulta para buscar o *Tipo* funcionário foi montada de modo a buscar apenas os campos Nome e Email. O campo **Resultado** é responsável por capturar as informações de resposta das requisições. Nesse campo, é usando o *Result in Table* que armazena informações como tempo de resposta, quantidade de bytes enviado, quantidade de bytes recebido, latência e informações adicionais que não serão abordadas nessa pesquisa.

4.4.3 Teste com múltiplas entidades

O terceiro cenário de teste consiste em verificar e comparar o comportamento das APIs para o cenário em que a API REST precisa executar múltiplas requisições ao servidor para montar a mesma estrutura que o GraphQL consegue montar em apenas uma chamada.

O objetivo desse teste é usar da habilidade de conectar diferentes entidades que o GraphQL possui por meio de consultas alinhadas para entender se haverá uma melhora em sua performance perante a API REST que, para buscar os mesmos dados, incorre no problema de N+1 requisições. Para esse teste é feito a busca dos dados dos funcionários junto com os dados do seu cargo, para isso a API REST irá usar dois *endpoints*, */funcionarios* e */cargo*. Já a API GraphQL usará apenas o *Tipo Funcionario* e alinhar internamente sua consulta com o *Tipo Cargo*. O Jmeter é configurado para elaborar as requisições e a Figura 4.5 ilustra a árvore de teste modelada.

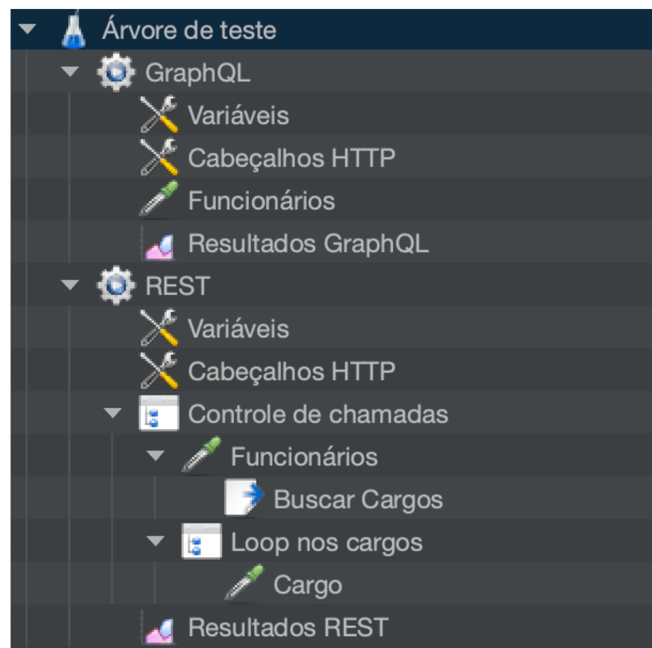

Figura 4.5: Árvore do terceiro teste.

Seguindo o padrão, cada API é desenhada em *Threads* diferentes para facilitar testes em apenas uma das APIs de forma isolada. O campo **Variáveis** e **Cabeçalhos HTTP** são campos comuns para as duas APIs e são responsáveis por definir os atributos e cabeçalhos necessários para que a requisição aconteça.

A *Thread* da API GraphQL possui mais dois campos. O primeiro, **"Funcionários"**, configura a requisição que será feita, em que define-se os campos que devem ser chamados usando a consulta alinhada para buscar todos os dados propostos pelo teste. O segundo campo é o **"Resultados"** que armazena informações como tempo de resposta, quantidade de bytes enviado, quantidade de bytes recebido e latência.

Já na *Thread* da API REST, o primeiro campo que se destaca é o **"Controle de chamadas"**, que é usado para agrupar em um único ambiente todas as chamadas que serão feitas. Isso é importante no momento de apresentar o resultado, pois as informações serão calculadas somando-se todas as requisições como se fosse uma única requisição. A primeira requisição feita é indicada pelo campo **Funcionários**, que faz uma requisição ao *endpoint /funcionarios* usando o método GET e recebe como retorno um vetor de funcionários em que cada funcionário possui o ID de um cargo. Para recuperar todos os IDs foi utilizado o campo **Buscar Cargos**, cuja função é armazenar em um vetor apenas os IDs que deve-se buscar para que os próximos campos **Loop nos cargos** e **Cargo** consigam percorrer por esse vetor e, para cada posição, buscar o cargo referente ao ID encontrado. Por fim, o campo **Resultado** é responsável por capturar informações como tempo de resposta, quantidade de bytes enviado, quantidade de bytes recebido e latência.

Capítulo 5

Resultados

Este capítulo descreve como foram feitos os testes, seus resultados e conclusões. A metodologia utilizada funcionou para as duas primeiras perguntas mencionadas no objetivo. Por conta da última pergunta exigir muito da capacidade de CPU do computador que estava sendo usado, foi necessário anular o cenário que apresentava um volume de dados maior no banco, isso não prejudicou o teste pois os dados do cenário com um número menor de dados foram coletados e foram suficientes para elaborar as conclusões. O objetivo principal foi alcançado, definiu-se as qualidades e defeitos de cada uma das arquiteturas utilizando três métricas principais: tempo de resposta da requisição, uso de CPU e consumo de memória.

5.1 Especificação dos Testes

Para a realização dos testes foi utilizado uma máquina com sistema operacional Linux com a distribuição Ubuntu versão 18.04, ela possui 16 Gb de memória RAM e um processador Intel Core i7 décima geração.

Busca-se uma comparação entre as arquiteturas REST e GraphQL em diferentes cenários. Cada teste terá dois volumes de dados: o primeiro cenário com 100 funcionários e o segundo cenário com 1000 funcionários.

Para o primeiro cenário, com 100 funcionários, foi simulado 1000 requisições ao longo de 10 segundos. Já no segundo cenário, com 1000 funcionários, realizou-se uma simulação de 1000 requisições por 60 segundos. Esse intervalo de tempo diferente nos dois cenários se fez necessário, pois o computador em que os testes estavam sendo feitos utilizava o máximo de sua capacidade de CPU ao aumentar a quantidade de dados mantendo-se o tempo fixo. Como consequência, isso aumentava a latência entre as requisições e afetava o progresso dos testes. Para evitar isso, o segundo cenário que possui maior quantidade

de dados possui também um intervalo de tempo maior que o primeiro e, assim, evita um uso excessivo da CPU.

Em todos os testes, foram coletados três variáveis que serão analisadas e comparadas: o tempo de resposta da requisição, a porcentagem de CPU usada pelo servidor e a porcentagem de memória usada pelo servidor.

5.2 Teste com Retornos Iguais

Para esse teste cada requisição buscava as informações de todos os funcionários de forma que as duas arquiteturas retornassem os mesmos dados.

5.2.1 Amostra de 100 funcionários

Para uma amostra de 100 funcionários no banco de dados, a Figura 5.1 ilustra uma comparação do tempo de resposta das duas arquiteturas. Isso indica que a arquitetura GraphQL possui um tempo de resposta maior para as primeiras requisições e demora um pouco mais que a arquitetura REST para se estabilizar. Essa estabilização, que na arquitetura GraphQL começa a partir da requisição 63, e na arquitetura REST na requisição 42, se deve ao fato do sistema elaborar um armazenamento em cache.

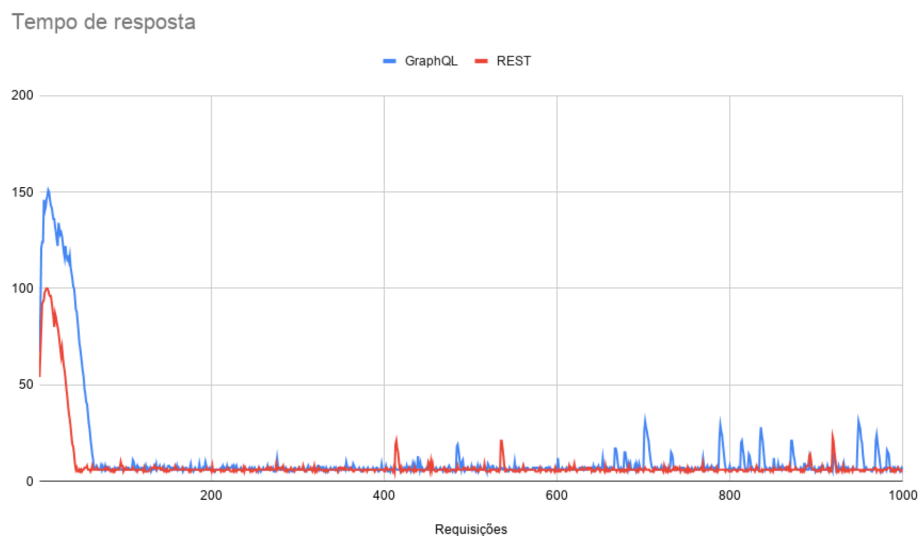

Figura 5.1: Tempo de resposta para teste com retornos iguais em amostra de 100 funcionários.

Percebe-se também que a partir do momento em que se estabiliza o tempo de resposta, as duas arquiteturas se comportam de maneira muito semelhante, com uma pequena diferença que na arquitetura GraphQL tem-se uma quantidade de picos maiores. A média

do tempo de resposta da arquitetura GraphQL ficou em 13,13 milissegundos enquanto que a média da arquitetura REST permaneceu 8,72 milissegundos. Já o desvio padrão na arquitetura GraphQL ficou em 24,84 milissegundos enquanto que na REST ficou em 13,65 milissegundos, um número maior na arquitetura GraphQL devido a maior quantidade de picos mencionada.

A Figura 5.2 ilustra a métrica de CPU, em que é possível observar que a arquitetura REST possui um consumo de CPU menor na maior parte das requisições, significando que a arquitetura GraphQL possui mais processamento interno pelo fato de fazer a filtragem dos campos de resposta, na média a arquitetura REST usou 61,21% de CPU enquanto a arquitetura GraphQL usou 73,76%.

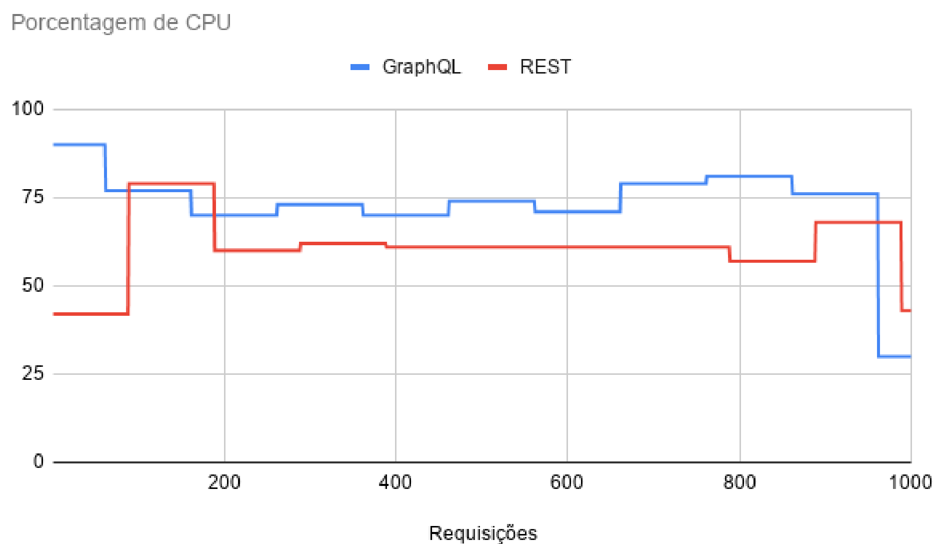

Figura 5.2: Consumo de CPU para teste com retornos iguais em amostra de 100 funcionários.

Para a métrica de memória, o gráfico da Figura 5.3 demonstra a comparação de consumo das duas arquiteturas. A arquitetura REST novamente leva vantagem nessa métrica, porém de forma sutil, uma vez que a média de consumo da API REST foi 0,55 enquanto que na API GraphQL foi de 0,63.

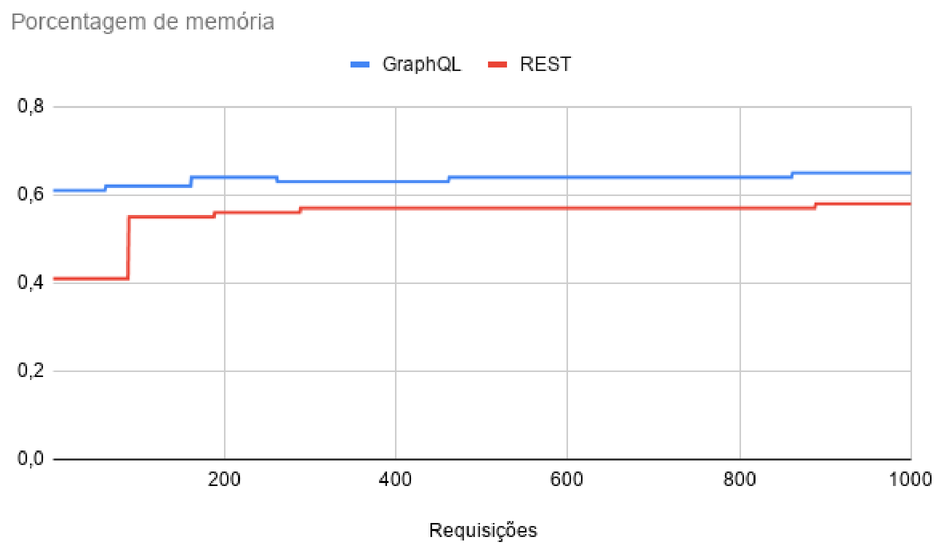

Figura 5.3: Consumo de memória para teste com retornos iguais em amostra de 100 funcionários.

5.2.2 Amostra de 1000 funcionários

Para essa segunda amostra, com 1000 funcionários no banco de dados, tem-se como objetivo verificar se o comportamento se assemelha ao cenário anterior já que agora existem mais dados para serem consumidos. Para a primeira métrica, tempo de resposta da requisição, elaborou-se o gráfico representado pela Figura 5.4.

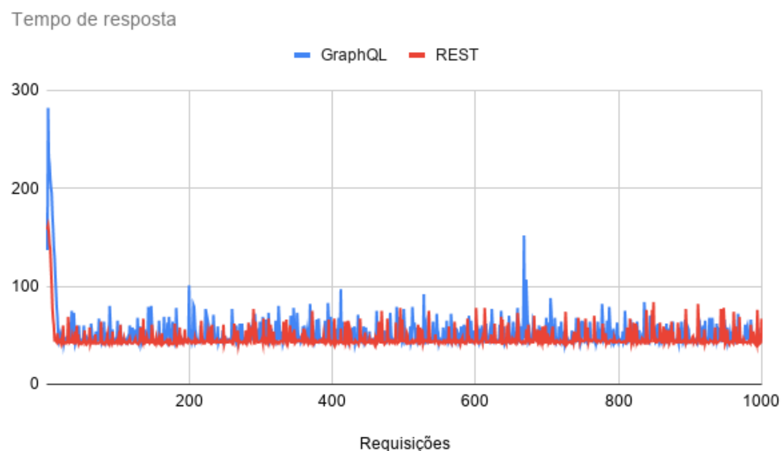

Figura 5.4: Tempo de resposta para teste com retornos iguais em amostra de 1000 funcionários.

Pode-se observar um comportamento análogo ao teste feito com menos dados, em que, inicialmente, a arquitetura GraphQL possui um pico maior do que a arquitetura REST e sua estabilização leva um pouco mais de tempo.

Nessa amostra, fica nítido que, mesmo o tempo de resposta sendo similar, as requisições GraphQL possuem mais picos, além de serem maiores durante o teste. No final, a média de resposta para a arquitetura GraphQL foi de 51,91 milissegundos, enquanto que na arquitetura REST foi de 47,17 milissegundos, o que é uma diferença pequena.

Na segunda métrica, consumo de CPU, elaborou-se o gráfico mostrado pela Figura 5.5.

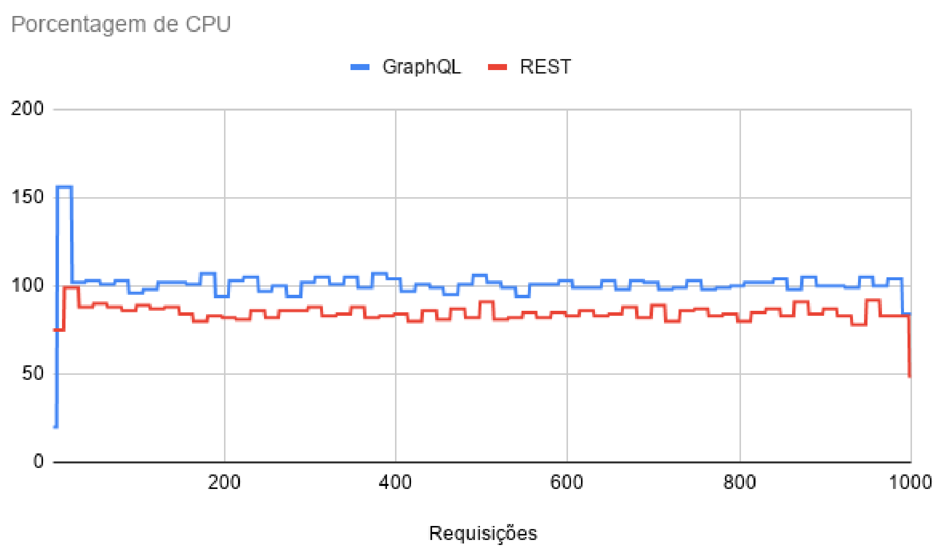

Figura 5.5: Consumo de CPU para teste com retornos iguais em amostra de 1000 funcionários.

Novamente seu comportamento é muito próximo ao teste com menos dados, em que a arquitetura REST utiliza menos CPU a todo momento. Um detalhe relevante se refere ao pico de consumo de CPU, em que a arquitetura GraphQL passou de 100%, o que significa que se fez necessário a utilização de mais de um core da CPU. A média de consumo feito pela API GraphQL ficou em 101,20%, enquanto que na API REST ficou em 84,66%. Por fim, a métrica de consumo de memória é mostrada na Figura 5.6.

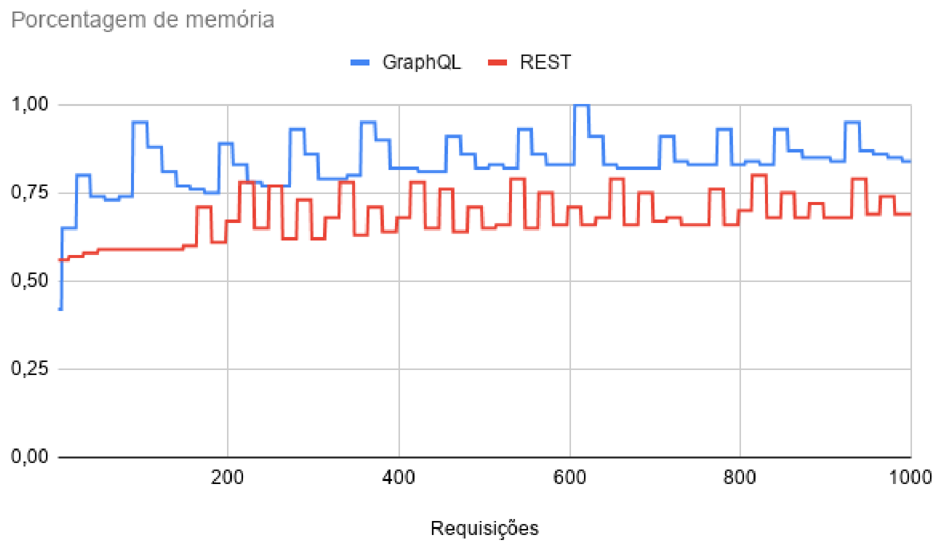

Figura 5.6: Consumo de memória para teste com retornos iguais em amostra de 1000 funcionários.

É possível notar uma maior variação do consumo de memória do que no cenário com menos carga de dados, mas, no geral, o comportamento se manteve. A API GraphQL apresentou um consumo de memória levemente superior comparado com a API REST.

Portanto, para esse primeiro teste, a arquitetura REST obteve um desempenho superior em relação à API GraphQL nas três métricas. Isso era esperado, pois a arquitetura GraphQL é mais complexa e, nesse primeiro cenário, não é considerado nenhuma vantagem oferecida por ela, o que a fez funcionar basicamente como uma arquitetura REST.

5.3 Teste com Retornos Diferentes

Para esse teste, pretende-se verificar a vantagem trazida pela arquitetura GraphQL utilizando a filtragem dos campos. Nesse cenário, as informações buscadas pela API GraphQL possuem menos campos que as informações provindas da API REST.

5.3.1 Amostra de 100 funcionários

Para uma amostra de 100 funcionários no banco de dados, a Figura 5.7 ilustra uma comparação do tempo de resposta das duas arquiteturas.

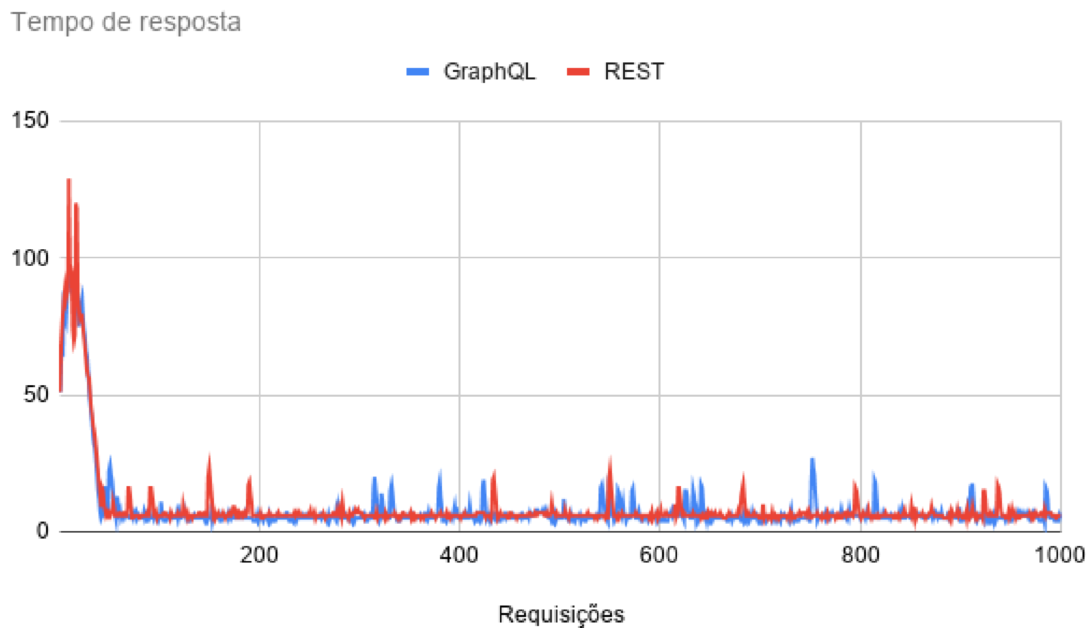

Figura 5.7: Tempo de resposta para teste com retornos diferentes em amostra de 100 funcionários.

Pelo gráfico, nota-se uma vantagem inicial da arquitetura GraphQL que possui um pico inicial menor do que a arquitetura REST. A estabilização do tempo de resposta ocorre quase que de forma idêntica e, diferente do primeiro teste, em que os picos ao longo do tempo são muito próximos. Na média, o tempo de resposta da arquitetura GraphQL ficou em 8,41 milissegundos, enquanto que na arquitetura REST em 9,07 milissegundos.

Com relação ao aspecto de uso de CPU, verifica-se que as duas arquiteturas se comportaram de forma similar, como mostra a Figura 5.8. Na média, a arquitetura GraphQL usou 59,10%, enquanto que a REST consumiu 62,70%.

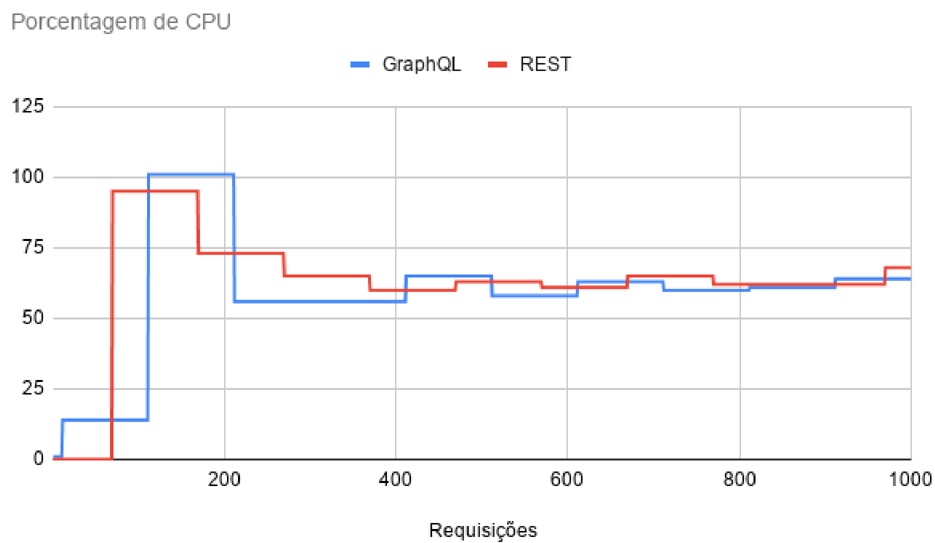

Figura 5.8: Consumo de CPU para teste com retornos diferentes em amostra de 100 funcionários.

No último aspecto de comparação, representado pela Figura 5.9, o consumo de memória se manteve maior na API GraphQL, mas com uma média próxima de 0,60% de consumo perante 0,53% da API REST.

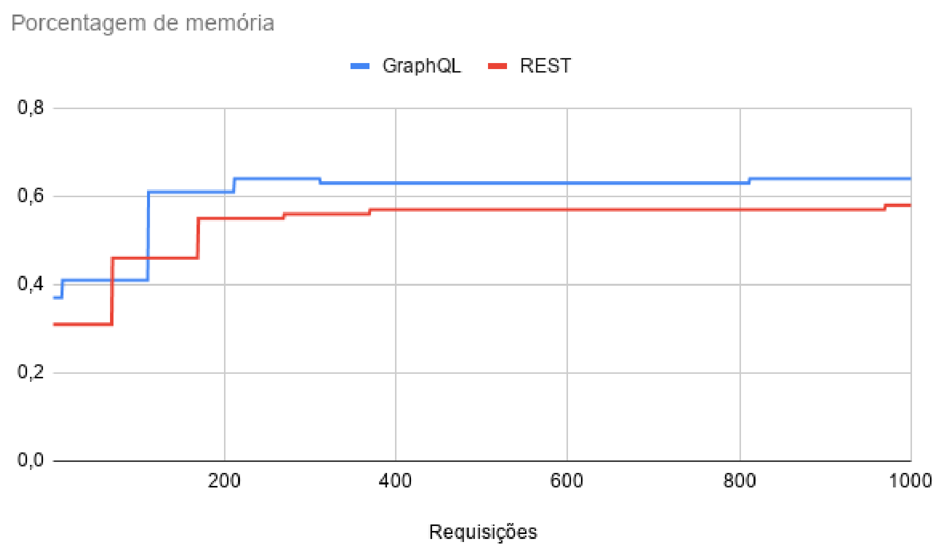

Figura 5.9: Consumo de memória para teste com retornos diferentes em amostra de 100 funcionários.

5.3.2 Amostra de 1000 funcionários

Considerando 1000 funcionários no banco de dados, constrói-se o gráfico baseado na métrica de tempo de resposta mostrado pela Figura 5.10. É possível observar que a API GraphQL possui uma vantagem ao longo de todo o teste e, assim como no primeiro cenário com 100 funcionários, seu pico inicial também é menor. Na média, o tempo de resposta na arquitetura GraphQL foi de 39,47 milissegundos contra uma média de 47,17 milissegundos da arquitetura REST.

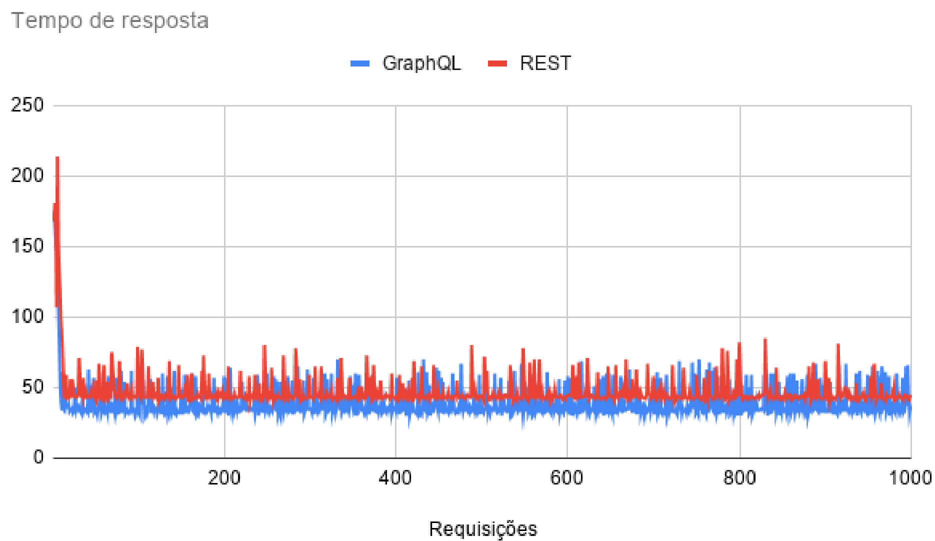

Figura 5.10: Tempo de resposta para teste com retornos diferentes em amostra de 1000 funcionários.

A Figura 5.11 apresenta um gráfico comparando as métricas de uso de CPU entre as duas APIs. Com ele é possível observar um igualdade no uso de CPU das duas arquiteturas, uma vez que, na média, a API GraphQL usou um pouco menos com 80,46% enquanto que a API REST usou 83,02%.

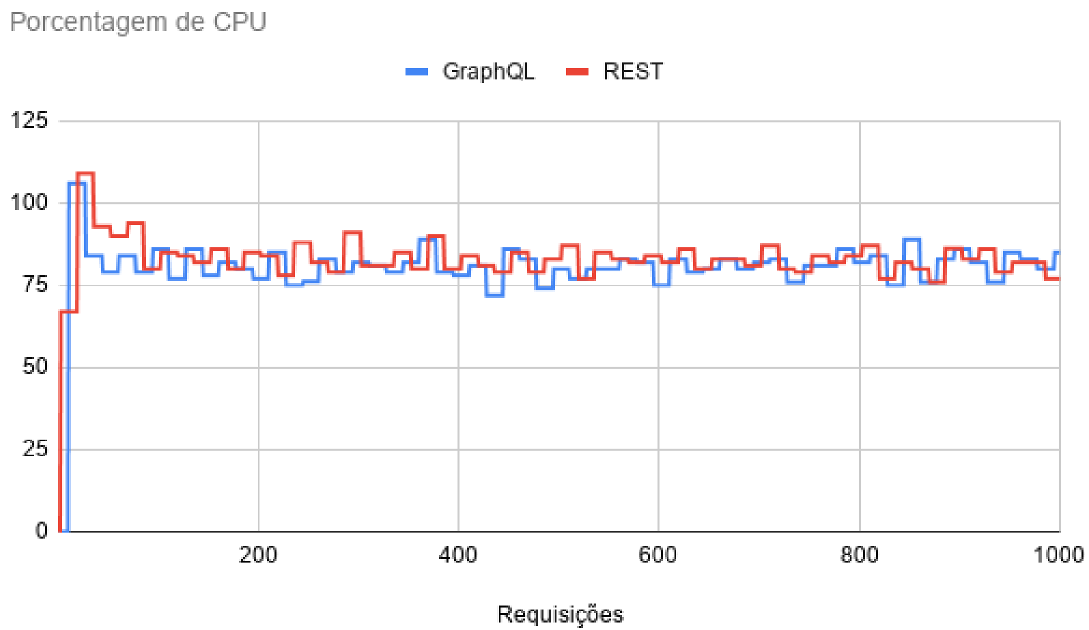

Figura 5.11: Consumo de CPU para teste com retornos diferentes em amostra de 1000 funcionários.

Por fim, a Figura 5.12 demonstra o consumo de memória. Pode-se verificar um maior variação no consumo de memória por parte da API REST ao longo do tempo e também uma pequena vantagem da API GraphQL por um consumo menor de memória. Em média, as duas APIs obtiveram um padrão similar nesse quesito: a API GraphQL com 0,63% e a API REST com 0,69%.

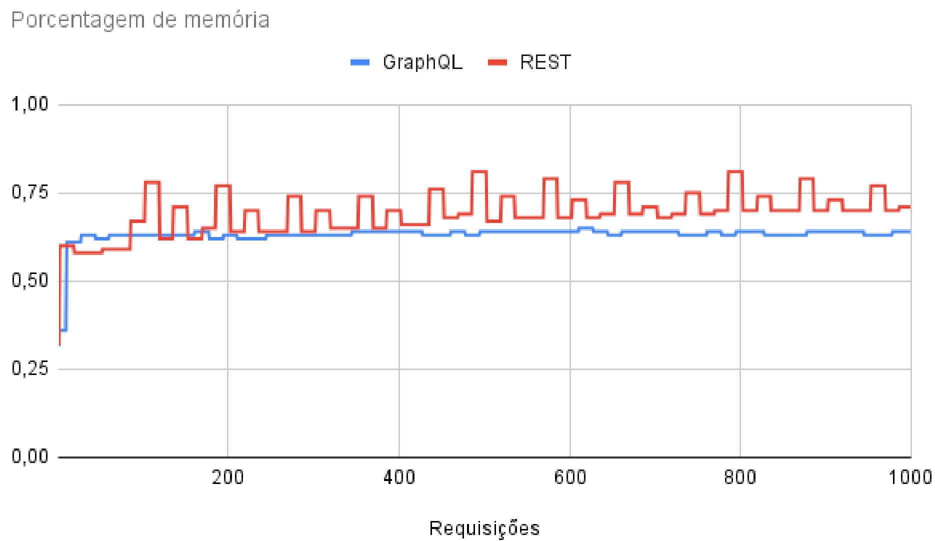

Figura 5.12: Consumo de memória para teste com retornos diferentes em amostra de 1000 funcionários.

Portanto, para esse segundo teste, a arquitetura GraphQL obteve um melhor desempenho, o que demonstra que, para consumo de dados filtrados, a API GraphQL leva vantagem. Acredita-se que a vantagem seria ainda maior se o teste fosse realizado com o ambiente do cliente sendo diferente do ambiente do servidor, como ocorre nos serviços em geral. Considerando a latência da rede somado ao fato da arquitetura GraphQL transmitir menos dados, o tempo de resposta das requisições seria menor em relação à arquitetura REST.

5.4 Teste com Múltiplas Entidades

Para esse teste, procura-se verificar qual a vantagem trazida pela arquitetura GraphQL considerando o alinhamento dos campos. Nesse cenário, as informações buscadas pela API GraphQL são acessíveis em apenas uma requisição, enquanto que para obter as mesmas informações, a API REST deve fazer uma quantidade de requisições proporcional a quantidade de funcionários.

Pelo fato desse teste ser robusto e complexo, o primeiro cenário, com 100 funcionários, se fez necessário um aumento no tempo da requisição, passando de 10 segundos para 60 segundos. Já o segundo cenário, com 1000 funcionários, não foi possível de ser concluído com base nas configurações e características do computador empregado no experimento, devido ao alto uso de processamento.

5.4.1 Amostra de 100 funcionários

Para a primeira métrica de avaliação, tempo de resposta, o gráfico da Figura 5.13 é elaborado. É evidente o ganho de performance da arquitetura GraphQL nesse cenário, pois o pico inicial da API REST é 1700 milissegundos, maior do que o pico da API GraphQL, e por meio da média, a comparação fica ainda mais clara. Na arquitetura GraphQL, obteve-se uma média de 39,34 milissegundos enquanto que na arquitetura REST 299,91 milissegundos, uma diferença expressiva de 662%.

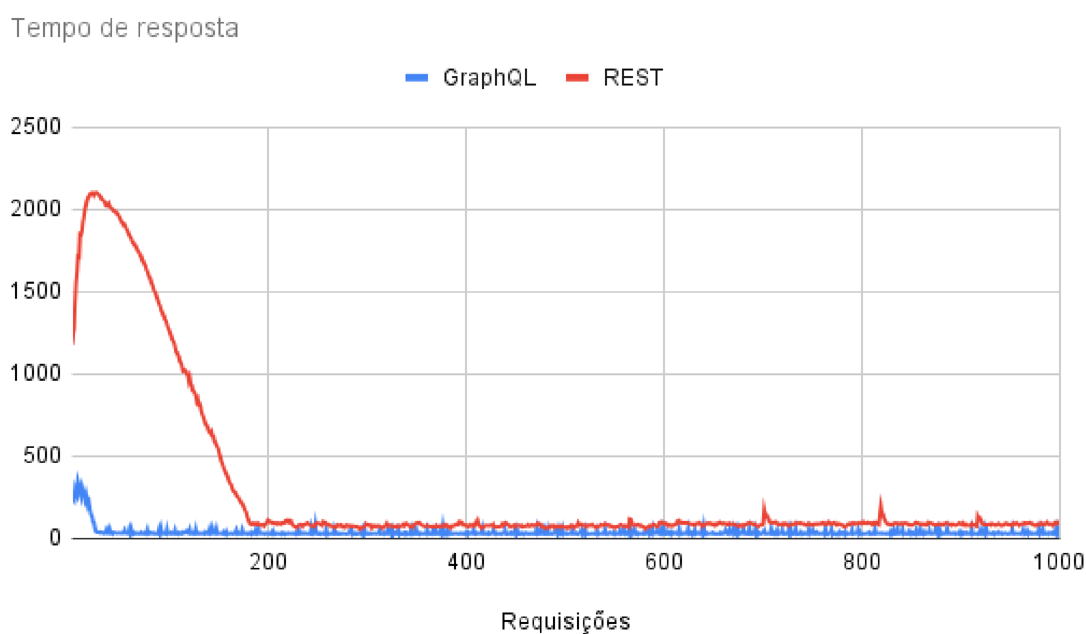

Figura 5.13: Tempo de resposta para teste com múltiplas entidades em amostra de 100 funcionários.

Em termos de uso de CPU, o gráfico da Figura 5.14 ilustra uma vantagem da arquitetura GraphQL, com média em 71,87% contra 94,55% da arquitetura REST.

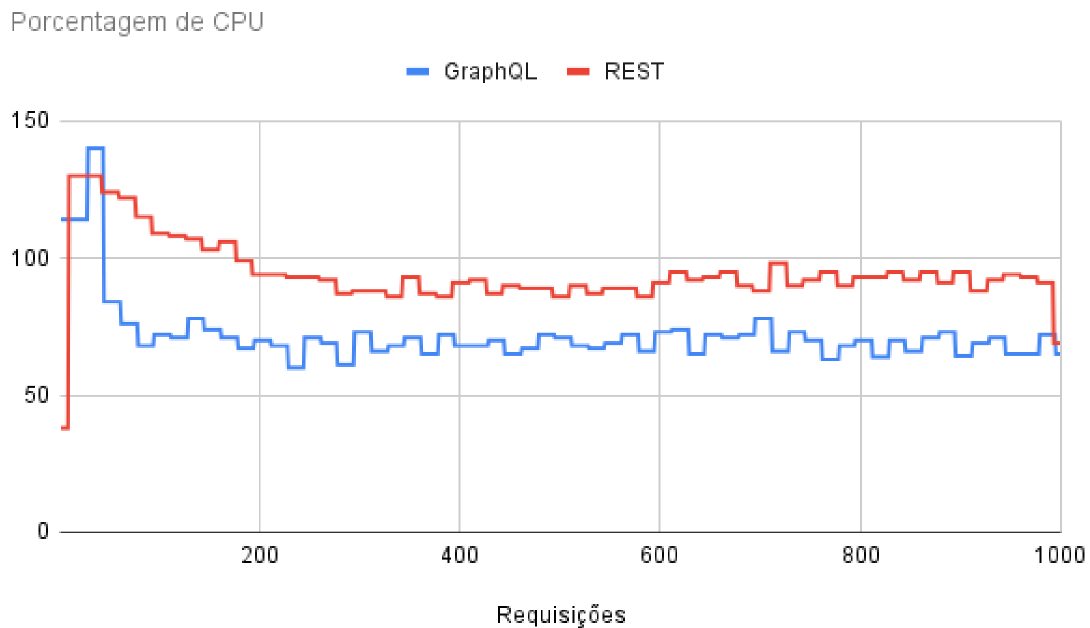

Figura 5.14: Consumo de CPU para teste com múltiplas entidades em amostra de 100 funcionários.

Na última métrica de análise, elaborou-se a Figura 5.15, com os dados do consumo de memória. É possível observar que o consumo de memória da API GraphQL foi maior em todos os momentos com uma média de 0,87% contra 0,72% da API REST. Por fim, é nítida a vantagem que a API GraphQL possui quando as duas APIs são submetidas a consultas alinhadas. Devido ao fato de ser possível elaborar toda a resposta do lado do servidor, seu tempo de resposta é menor com, inclusive, menos gasto de CPU. Sua vantagem seria ainda mais evidenciada caso fosse simulando um sistema real com cliente e servidor em máquinas diferentes, em que as informações estariam trafegando na rede e aumentando a latência das múltiplas requisições da API REST.

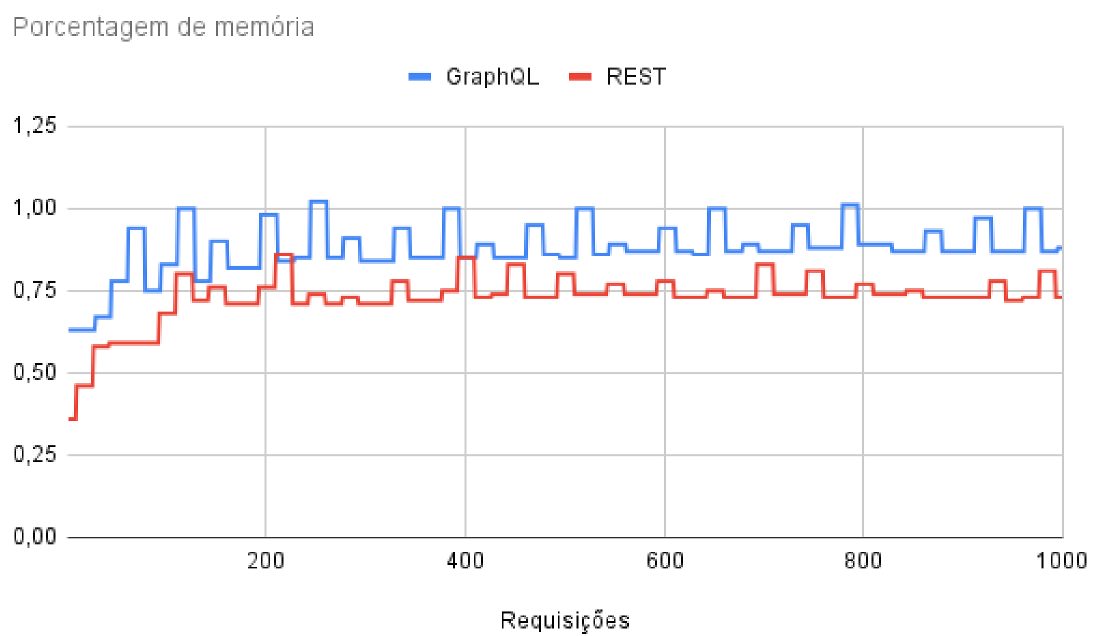

Figura 5.15: Consumo de memória para teste com múltiplas entidades em amostra de 100 funcionários.

Capítulo 6

Conclusão

6.1 Considerações Finais

Este trabalho apresentou diferentes cenários para uma comparação de performance entre as arquiteturas GraphQL e REST. O objetivo principal consistiu em trazer as vantagens propostas por cada arquitetura e verificar em quais cenários uma apresenta vantagens em relação a outra. O primeiro cenário analisou a vantagem de realizar um cache rápido da informações oferecido pela arquitetura REST. O segundo cenário explorou a vantagem de buscar menos campos proposto pelo modelo GraphQL. Por fim, o último cenário abordou a maior vantagem proposta pelo arquitetura GraphQL, consultar múltiplas entidades em uma única requisição por meio de uma consulta alinhada. Cada cenário realizou testes com dois volumes de dados diferentes para evitar a ocorrência de viés nos testes, sabendo-se da quantidade de dados trafegados durante os experimentos. O banco de dados foi construído ao executar algoritmos aleatórios, garantindo-se que em todos os testes, a quantidade de consultas feitas no banco de dados eram as mesmas para as duas APIs.

Concluiu-se que, em ambientes em que é necessário uma flexibilidade na busca dos dados, a arquitetura GraphQL oferece uma grande vantagem de performance, principalmente se nessa busca houver a necessidade de se obter diferentes entidades. Por outro lado, em cenários em que a construção da API é feita de forma exclusiva para uma única aplicação, a API REST leva vantagem. Por conta da API ser construída para uma aplicação específica, não é possível elaborar os *endpoints* de forma a retornar exatamente os dados que serão empregados, evitando-se o problema de *Overfetching* e *Underfetching* da arquitetura REST.

6.2 Trabalhos Futuros

GraphQL é um arquitetura muito recente e, por isso, ainda existem muitos detalhes a ser levado em consideração em uma pesquisa. Para essa monografia, é possível coletar mais métricas de performance para realizar uma comparação mais profunda. Além disso, elaborar diferentes cenários de banco de dados para entender como as APIs de comportam com entidades diferentes. Uma limitação dessa monografia foi utilizar apenas uma linguagem de programação para elaborar as duas arquiteturas. Portanto, uma melhoria seria implementar as APIs usando diferentes linguagens para tirar o viés da tecnologia.

Por fim, mais uma pesquisa foi feita para contribuir com a literatura relacionada ao modelo GraphQL, algo ainda muito recente, mas que tem se popularizado nos últimos anos.

Referências

[1] Johnson, Joseph: *Number of internet users worldwide from 2005 to 2019*. https://www.statista.com/statistics/273018/number-of-internet-users-worldwide/, acesso em 2020-02-27. 1

[2] Nadareishvili, Irakli, Ronnie Mitra, Matt McLarty e Mike Amundsen: *Microservice Architecture: Aligning Principles, Practices, and Culture*. O'Reilly Media, Inc., 1st edição, 2016, ISBN 1491956259. 2

[3] Fielding, Roy Thomas: *Architectural Styles and the Design of Network-based Software Architectures*. Tese de Doutoramento, University of California, Irvine, 2000. 2

[4] *Ebay aims to be 'operating system' for all e-commerce on internet*. https://www.wsj.com/articles/SB974675427606513763. 3

[5] *Amazon.com launches web services; developers can now incorporate amazon.com content and features into their own web sites; extends "welcome mat" for developers*. https://press.aboutamazon.com/news-releases/news-release-details/amazoncom-launches-web-services. 3

[6] *Facebook web site*. https://www.facebook.com/. 3

[7] *Graphql: A data query language*, 2015. https://engineering.fb.com/2015/09/14/core-data/graphql-a-data-query-language/. 3

[8] *Facebook's graphql gets its own open-source foundation*, 2018. https://techcrunch.com/2018/11/06/facebooks-graphql-gets-its-own-open-source-foundation/. 3

[9] *Evolution of http*. https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/Evolution_of_HTTP. 5

[10] https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/GET. 6

[11] https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/POST. 6

[12] https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers. 6

[13] *Introdução ao json*. https://www.json.org/json-pt.html. 7

[14] *Graphql - a query language for your api*. https://graphql.org/. 8

[15] https://www.apollographql.com/blog/4-simple-ways-to-call-a-graphql-api-a6807bcdb3
8

[16] *Serving over http*. https://graphql.org/learn/serving-over-http/. 8

[17] *list of directories of content types and subtypes*. http://www.iana.org/assignments/media-types/media-types.xhtml. 8

[18] *Graphql post request*. 8

[19] *Graphl queries and mutations*. https://graphql.org/learn/queries/. 10

[20] *Get message body*. https://tools.ietf.org/html/rfc2616#section-4.3. 19

[21] *Http response status codes*. https://developer.mozilla.org/en-US/docs/Web/HTTP/Status. 21

[22] Eizinger, Thomas: *Api design in distributed systems: a comparison between graphql and rest*, 2017. 23

[23] Oggier, Camille: *How fast graphql is compared to rest apis*. 2020. 23

[24] *React web site*. https://reactjs.org/. 23

[25] Landeiro, Mafalda Isabel Ferreira: *Analysis of GraphQL performance: a case study*. Tese de Doutoramento, 2019. 24