# GraphQL - Rest in Peace: Uma Análise Comparativa

**Raphael de Oliveira Tavares[1], Sandro Roberto Fernandes[2]**

[1]Instituto Federal de Educação, Ciência e Tecnologia do Sudeste de Minas Gerais – *Campus* Juiz de Fora

[2]Núcleo de Informática - Instituto Federal de Educação, Ciência e Tecnologia do Sudeste de Minas Gerais – *Campus* Juiz de Fora

raphael_tavares2000@hotmail.com, sandro.fernandes@ifsudestemg.edu.br

**Abstract.** *This project aimed to develop applications that use the REST architecture and the query language GraphQL, aiming at a study of performances and implementations. The conclusions show that GraphQL has a better performance than REST, when compared in relation to response time. REST, on the other hand, presents a lower implementation effort. This work presents a careful evaluation for each technology, besides comparing them, and the result is conclusive to determine which would be more beneficial to the context of the target company of the study.*

**Resumo.** *Esse projeto visou desenvolver aplicações que utilizassem a arquitetura REST e a query language GraphQL, visando um estudo de performances e implementações. As conclusões mostram que o GraphQL possui uma performance superior ao REST, quando comparados em relação ao tempo de resposta. Já o REST, apresenta um esforço de implementação menor. Este trabalho apresenta uma avaliação criteriosa para cada tecnologia, além de compará-las, sendo o resultado conclusivo para determinar qual seria mais benéfica para o contexto da empresa alvo do estudo.*

## 1. Introdução

*A arquitetura* REST (*Representational State Transfer* - Transferência de Estado Representacional) é um estilo de arquitetura de software com suas convenções para facilitar a criação, utilização e transferência de dados entre webservices. A tecnologia foi criada em 2001, sendo considerada antiga e é bem estabilizada no mercado. A mesma tem seus benefícios, mas muitas vezes é utilizada apenas por já ser conhecida pelos programadores e não por seus benefícios reais.

*GraphQL* é uma tecnologia criada em 2012 pelo *Facebook* e lançada publicamente em 2015. Inicialmente, essa linguagem de consulta de dados foi criada para resolver problemas internos ao *Facebook*, como a alta taxa de tráfego de dados desnecessário através das *requests*, problema proveniente da arquitetura *REST*. Após seu

lançamento, a tecnologia se popularizou e evoluiu, mas a mesma ainda é muito recente e há muito o que ser definido em termos de sua qualidade, performance e usabilidade.

Para definir quais casos de uso são mais adequados à implementação do *GraphQL* e quais se encaixam melhor no *REST*, esse trabalho visa implementar ambas as tecnologias em diferentes cenários e comparar suas performances.

## 2. Revisão Sistemática

Foi realizada uma revisão sistemática de artigos científicos na base acadêmica Capes, utilizando a língua inglesa e as *strings* de busca utilizadas, após alterações necessárias para se obter resultados esperados, foram:

- "*rest api*" AND "comparative" AND "*graphql*"
- "*rest api*" AND "performance" AND "*graphql*"
- "*graphql*" AND ("*design*" ("*pattern*" OR "*patterns*") OR "*good practices*")

O *GraphQL* se mostrou ser uma tecnologia com escassez de material científico, provavelmente por ter sido disponibilizada há poucos anos. em virtude desta falta de referências bibliográficas relevantes, as *strings* de busca foram alteradas, tornando-se mais abrangentes e retornando, assim, mais resultados para a pesquisa:

- ("*graphql*") AND ("*rest*")
- ("*graphql*") AND ("*design pattern*")
- ("*graphql*")

Com essas *strings* de busca, foram encontrados 65, 104 e 163 artigos, respectivamente, resultando em um total de 332 artigos. Desses 332 artigos, todos foram analisados a partir de seu título e *abstract*, para que houvesse um filtro dos mais relevantes, que foram lidos integralmente, para enfim, selecionar os 9 artigos mais relevantes para a pesquisa.

Em Gleison Brito (2015) é proposto um estudo com estudantes de graduação e já graduados, que tinham, ou experiencia com *REST* e *GraphQL*, ou só experiência com *REST*. Após o estudo, foi constatado empiricamente que mesmo entre os alunos com experiência prévia em *REST* e sem experiência em *GraphQL*, o segundo grupo obtinha vantagem em relação ao tempo de implementação. O estudo foi realizado utilizando a linguagem de programação Python.

Já em Gleison Brito *et. al.* (2019), os autores buscaram definir qual era a performance do *GraphQL* em comparação ao *REST*. Através do estudo de artigos de blogs e da migração de serviços *REST* para *GraphQL*, puderam concluir que com o *GraphQL*, há uma redução de em média 94% no tamanho do corpo da *response* de

uma *request HTTP*, o que resulta em uma redução significativa na largura de banda utilizada.

No trabalho de Alan Cha *et. al.* (2020) é proposto um medidor de custo de *queries GraphQL*. O artigo clama ser o primeiro medidor a conseguir medir com boa acurácia, definir convenções semânticas do *GraphQL* e também medidas de complexidade da *query*.

Devido à natureza de retornar dados aninhados do *GraphQL*, o custo de suas queries pode aumentar exponencialmente. Para isso, é proposto um trabalho em Georgios Mavroudeas et al (2021) que resulta em uma inteligência artificial com *Machine Learning* capaz de calcular e prever o custo de uma *query* de qualquer projeto *GraphQL*.

Erik Wittern (2018), propõe um *GraphQL-Wrapper* (Embrulho-GraphQL) para *API's REST*, que implementa uma camada adicional em *API's REST* para que essas disponham dos benefícios do *GraphQL*, que são o fato de trafegar menos informações pela rede, requerendo menos banda larga e também flexibilizando mais as consultas para o *frontend*. O que o *wrapper* faz é receber uma requisição *GraphQL* e gerenciar os requisitos requisitados para buscá-los através dos *endpoints REST*.

O trabalho de Armin Lawi *et. al.* (2021) critica o fato de os estudos existentes serem realizados em ambientes e cenários muito mais simples do que a realidade, propondo então, para contrapor esse ônus, um estudo em um ambiente real, o sistema de gerenciamento de informações da *Hasanuddin University Research and Community Service Institute*, chamado SIM-LP2M. O trabalho conclui que o *GraphQL* possui um maior tempo de resposta em comparação ao *REST*. Conclui também que o *GraphQL* tem um *throughput*, que é a capacidade de lidar com quantidades de *requests*, menor que o do *REST*. Das definições feitas por esse artigo, as únicas em que mostraram pontos positivos em não utilizar o *REST* seriam os fatos de o *GraphQL* utilizar uma porcentagem menor de CPU e memória para realizar seus processamentos.

Em Erik Wittern *et. al.* (2019) são estudadas boas práticas para o desenvolvimento de *schemas GraphQL*. O resultado é conclusivo, descobrindo convenções utilizadas no desenvolvimento e brechas de desempenho nas aplicações.

Li Li *et. al.* (2015) apresenta um *framework* e *design patterns* específicos para o desenvolvimento de *API's REST* que permitem que as mesmas sejam extensíveis, podendo comunicar-se com outros sistemas e até aplicar mudanças sem quebrar os sistemas já integrados a mesma.

No trabalho de Hayet Brabra *et. al.* (2018) é proposto um algoritmo para detecção de boas práticas (*patterns*) e más práticas (*anti patterns*) de desenvolvimento de *API's REST Cloud* de acordo com a *Open Cloud Computing Interface* (OCCI), além de sugestões de correção para os *anti patterns*.

# 3. Fundamentação Teórica

## 3.1. *REST*

A *REST* é uma arquitetura de software que cria convenções para implementações de uma API e foi criada em no ano de 2000. Os princípios que uma API deve seguir para se encaixar na arquitetura *REST* e poder ser chamada de *RESTful* são os seguintes, de acordo com *AWS s.d.*:

- Interface Uniforme: A interface da *API* deve identificar recursos nas solicitações. Deve também prover informações suficientes para o cliente conseguir modificar ou excluir recursos. Os clientes devem receber informações auto descritivas que contêm metadados, informando ao mesmo como utilizar a interface. Por fim, os clientes também devem receber informações sobre todos os recursos relacionados ao que precisam para concluir uma tarefa, recebendo *hiperlinks* da interface para que os clientes consigam descobrir dinamicamente mais recursos.
- Ausência de estado: Cada solicitação do cliente deve ser completada independentemente de qualquer solicitação anterior. Isso significa que não é preciso ter nenhum estado armazenado para fazer os processamentos necessários, podendo, assim, ser chamada de *stateless*.
- Sistema em camadas: A interface poderá fazer a intermediação com outros servidores, acessando recursos externos. O serviço *RESTful* pode ser executado em diversas camadas, como uma camada de segurança, uma de aplicação e regras de negócio, todas trabalhando juntas para atender às solicitações do cliente.
- Capacidade de armazenamento: Os serviços *RESTful* devem ter a capacidade de armazenar recursos, seja em um banco de dados, em cache, em um serviço *cloud,* ou qualquer outro.
- Código sob demanda: O servidor pode estender a funcionalidade do cliente, passando códigos que façam validações, por exemplo.

## 3.2. *GraphQL*

O *GraphQL* é uma *query language* (linguagem de consulta) para *API's* e fornece uma descrição completa dos dados da *API*.

Esta *query language* permite que o usuário requisite dados específicos, não necessitando requisitar e trafegar dados inúteis, reduzindo assim, a banda larga utilizada para responder a requisições.

A mesma também possui a capacidade de agrupar os recursos requisitados em apenas uma *request*.

### 3.3. Gatling

O Gatling é uma ferramenta de testes de carga. Testes de carga são testes que realizam um número X de requisições para um servidor em um tempo Y para comprovar se o servidor suporta lidar com aquela carga.

Esse tipo de teste é importante para que seja possível antecipar a performance de uma aplicação, antes de colocá-la no ar para o uso, prevendo assim, desastres.

Como a ferramenta mede o tempo das requisições, também é possível utilizá-la para realizar testes de performance, que medem apenas a eficácia e eficiência de uma aplicação, sem utilizar grandes volumes de carga.

## 4. Metodologia

### 4.1. Ambiente

A maioria dos estudos existentes sobre *GraphQL* são realizados em ambientes e cenários muito simples e distantes da realidade (Armin Lawi *et. al.;*2021). Em virtude disso foi buscado uma forma de conseguir realizar um trabalho que representasse um cenário aproximado da realidade. Assim foi provido um ambiente da empresa BTG Pactual, que, em concordância com superiores, permitiu a utilização de seu ambiente e dados para o estudo. Parte desse projeto foi utilizado também como uma demanda interna da empresa, proporcionando um teste dentro de um cenário real.

Por conta da necessidade de manter os dados e informações da empresa ocultos, sempre que for exposto algum código ou citado algo sobre o contexto neste trabalho, nomes fictícios serão usados para o contexto, o nome do banco de dados, das tabelas, para seus dados, nome de variáveis, tipos e qualquer outra nomenclatura que possa comprometer o sigilo e segurança.

A demanda da empresa era fazer um estudo, denominado de "*spike*" em metodologias ágeis, para definir qual estrutura, REST ou *GraphQL*, seria mais benéfica de se implementar para suprir as necessidades de uma devida demanda. Tal demanda consistia em implementar um algoritmo que pudesse receber filtros dinamicamente, recebendo nenhum, um, ou mais filtros e retornasse registros do banco de dados filtrados a partir desses filtros recebidos como parâmetro. As tecnologias utilizadas foram C#, para a linguagem de programação e MySQL e PostgreSQL, para os bancos de dados. As mesmas foram escolhidas por serem utilizadas no contexto da empresa e, ou já existirem os recursos necessários, no caso dos bancos de dados.

### 4.2. Desenvolvimento de *endpoint REST*

O primeiro passo do estudo constituiu-se em implementar um *endpoint REST* em uma aplicação já existente. Para tal implementação, foi criada uma branch no repositório, chamada "feature/spike-contexto-fictício". Nessa *branch* foi programado um método que

recebia nenhum ou um parâmetro e decidia a partir do recebimento ou não do parâmetro se faria consulta em apenas uma tabela do banco ou se teria que fazer uma consulta em três tabelas, conectadas através do comando "*INNER JOIN*" de banco de dados. Aqui está o código original:

```
public class SpikeController : ApiController
{
    private readonly IContextoRepository _contextoRepository;

    0 references | 0 changes | 0 authors, 0 changes | 0 exceptions
    public SpikeController(IContextoRepository contextoRepository)
    {
        _contextoRepository = contextoRepository;
    }

    [Route("Spike/GetRegistrosSpike")]
    [HttpGet]
    0 references | 0 changes | 0 authors, 0 changes | 0 requests | 0 exceptions
    public HttpResponseMessage GetRegstrosSpike([FromUri] string filtro)
    {
        try
        {
            var response = _contextoRepository.GetRegistrosSpike(filtro);

            return Request.CreateResponse(HttpStatusCode.OK, response);
        }
        catch(Exception ex)
        {
            return Request.CreateResponse(HttpStatusCode.InternalServerError, ex.Message + ex.InnerException);
        }
    }
```

Figura 01 – Código do *Controller,* que é a camada mais externa da *API*, que recebe a requisição *HTTP* e gerencia o fluxo. Fonte: O Autor.

A camada de *controller* é acionada a partir da rota "Spike/GetRegistrosSpike" e aciona o método onde está contido o algoritmo que consulta o banco de dados "_contextoRepository.GetRegistrosSpike(filtro)", Figura 02 abaixo:

```
public List<RegistroSpike> GetRegistrosSpike(string filtro = "")
{
    string query = $"SELECT ta.*";
    string sqlWhere = " WHERE 1=1 AND (";
    if (!string.IsNullOrEmpty(filtro))
    {
        query += ", rca.*, tlp.ID as TLP_ID, tlp.COLUNA_2, tlp.COLUNA_3, tlp.COLUNA_4," +
            " tlp.COLUNA_5, tlp.COLUNA_6, tlp.COLUNA_7, tlp.COLUNA_8," +
            " tlp.COLUNA_9, tlp.COLUNA_10 as TLP_COLUNA_10" +
            " FROM Schema_Ficticio1.TABELA_1 as ta";

        query += " INNER JOIN Schema_Ficticio2.TABELA_1 rca on rca.TA_ID = ta.ID";
        query += " INNER JOIN Schema_Ficticio1.TABELA_2 tlp on ta.TLP_ID = tlp.TLP_ID";
        sqlWhere += "rca.COLUNA_1 IS NOT null";
        sqlWhere += $" AND tlp.COLUNA_2 = '{filtro}'";
    }
    sqlWhere += ");";

    query += sqlWhere;

    _dbContext.Database.CommandTimeout = 30000;

    return _dbContext.Database.SqlQuery<RegistroSpike>(query).ToList();
}
```

Figura 02 – Código do *Repository*. Fonte: O Autor.

A camada de model, com os arquivos “Contexto” e “RegistroSpike” possuem apenas definições do tipo dos dados.

### 4.3. Desenvolvimento de endpoint *GraphQL*

O segundo passo consistiu-se em desenvolver um endpoint *GraphQL* que também recebesse zero ou um parâmetro, e consultasse as mesmas tabelas do endpoint *REST*, para utilizar esse parâmetro como filtro, caso o mesmo fosse provido.

O projeto foi estruturado em um schema, onde são declarados os arquivos de *query* e *mutation*, que são onde ficam declarados os algoritmos de busca e alteração de dados no(s) banco(s) de dados, o model, que é onde fica definido o tipo de cada dado, o arquivo de conversão do model para o tipo do *GraphQL* e o arquivo de conexão com o banco de dados.

```
public class ContextoSchema : Schema
{
    1 reference | Raphael Tavares, 14 days ago | 1 author, 1 change
    public ContextoSchema(IServiceProvider serviceProvider) : base(serviceProvider)
    {
        Query = serviceProvider.GetRequiredService<ContextoQuery>();
        Mutation = serviceProvider.GetRequiredService<ContextoMutation>();
    }
}
```

Figura 03 – Código do *Schema*. Fonte: O Autor.

```
public class ContextoQuery : ObjectGraphType
{
    private readonly IDbConnection _dBConnection;

    0 references | 0 changes | 0 authors, 0 changes
    public ContextoQuery(IDbConnection DBConnection)
    {
        _dBConnection = DBConnection;



        Field<ListGraphType<ContextoType>>("GetRegistrosContexto",
            arguments: new QueryArguments(
                new QueryArgument<StringGraphType> { Name = "filtro" }
            ), resolve: context =>
            {
                var filtro = context.GetArgument<string>("filtro");

                if (filtro == null)
                {
                    throw new Exception("No filter especified");
                }

                var registrosContexto = _dBConnection.GetRegistrosContexto(filtro);

                return registrosContexto;
            });
```

Figura 04 –Arquivo de *Query*, onde são gerenciadas as consultas dos dados, utilizando o *repository*. Fonte: O Autor.

```
public class ContextoMutation : ObjectGraphType
{
    private readonly IDbConnection _dBConnection;

    0 references | Raphael Tavares, 14 days ago | 1 author, 1 change
    public ContextoMutation(IDbConnection DBConnection)
    {
        _dBConnection = DBConnection;

        Field<IntGraphType>(
            "MutationExample",
            arguments: new QueryArguments(
                new QueryArgument<NonNullGraphType<IntGraphType>> { Name = "ExampleParameter" }
                ),
            resolve: context =>
            {
                var exampleParameter = context.GetArgument<int>("ExampleParameter");
                return exampleParameter + 1;
            }
            );
    }
}
```

Figura 05 – Arquivo de *Mutation*, onde são gerenciadas as alterações nos dados, utilizando o *repository*. Fonte: O Autor.

```
2 references | 0 changes | 0 authors, 0 changes
public IEnumerable<Contexto> GetRegistrosContexto(string filtro)
{
    string query = $"SELECT ta.*";
    string sqlWhere = " WHERE 1=1 AND (";
    if (!string.IsNullOrEmpty(filtro))
    {
        query += ", rca.*, tlp.ID as TLP_ID, tlp.COLUNA2, tlp.COLUNA3, tlp.COLUNA4," +
            " tlp.COLUNA5, tlp.COLUNA6, tlp.COLUNA7, tlp.COLUNA8," +
            " tlp.COLUNA9, tlp.COLUNA10 as TLP_COLUNA10" +
            " FROM Schema_Ficticio1.TABELA1 as ta";

        query += " INNER JOIN Schema_Ficticio2.TABELA1 rca on rca.TA_ID = ta.ID";
        query += " INNER JOIN Schema_Ficticio1.TABELA2 tlp on ta.TLP_ID = tlp.TLP_ID";
        sqlWhere += "rca.COLUNA1 IS NOT null";
        sqlWhere += $" AND tlp.COLUNA9 = '{filtro}'";
    }
    sqlWhere += ");";

    query += sqlWhere;

    List<Contexto> list = new List<Contexto>();

    if (this.OpenConnection() == true)
    {
        MySqlCommand cmd = new MySqlCommand(query, connection);
        MySqlDataReader dataReader = cmd.ExecuteReader();

        list = DataReaderMapToList<Contexto>(dataReader);

        dataReader.Close();

        this.CloseConnection();

        return list;
    }
    else
    {
        return list;
    }
}
```

Figura 06 –Arquivo de *Repository*. Fonte: O Autor.

### 4.4. Testes

Para a realização de testes de carga, foi utilizado a ferramenta gatling, que permite gerar relatórios de métricas. Estes relatórios foram gerados para todos os *endpoints* desenvolvidos nesse trabalho. Um exemplo do código utilizado nesta ferramenta pode ser visto na Figura 07 abaixo:

```
package Contexto
import scala.concurrent.duration._
import io.gatling.core.Predef._
import io.gatling.http.Predef._
class CreateApplicationSimulation extends Simulation {
  val httpProtocol = http
    .userAgentHeader("Mozilla/5.0 (Windows NT 5.1; rv:31.0) Gecko/20100101 Firefox/31.0")
  val bodyJson = "query{ContextoTwoDb(filtro: 'valor'){iD}}"
  val scn = scenario("Create Application")
    .exec(
      http("graphql_contexto_db")
        .post("http://url.btg.com/graphql")
        .header("content-type", "application/graphql")
        .body(StringBody(bodyJson))
    )
  setUp(
    scn.inject(rampUsersPerSec(1).to(5).during(10.seconds))
  ).protocols(httpProtocol)
}
```

Figura 07 –Arquivo de teste de carga do gatling. Fonte: O Autor.

As alterações de código entre os *endpoint* foram a variável “bodyJson”, o nome dentro de http(“”), o método, podendo ser *GET* ou *POST* e a *URL* dentro do método (ex: post(“”)).

A penúltima linha de código serve para definir quantas *requests* serão feitas por segundo e durante quantos segundos. No exemplo, o programa começará fazendo 1 *request* e aumentará até fazer 5 *requests* por segundo, ao longo de 10 segundos.

### 4.5. Métricas de código

Utilizando o recurso de análise de código do *Visual Studio*, foram geradas métricas sobre o código, que permitem com que o mesmo seja analisado. As métricas geradas para análise foram as seguintes, de acordo com *Microsoft (2022)*:

- Índice de manutenção: Calcula a facilidade relativa de manter o código, representada em um valor de 0 a 100. Quanto mais alto o valor, significa melhor manutenção.
- Complexidade ciclomática: Mede a complexidade do código com base no número de caminhos de código diferentes no fluxo do programa. Um programa que tem um fluxo de controle complexo requer mais testes para obter uma boa cobertura de código e é menos mantenedível.
- Profundidade de Herança: Indica o número de classes diferentes que herdam umas das outras. Uma alteração em uma classe base pode afetar qualquer uma de

suas classes herdadas. Quanto maior esse número, maior a herança e maior o potencial de modificações de classe base para resultar em uma alteração significativa. Para Profundidade de Herança, um valor baixo é bom e um valor alto é ruim.

- Acoplamento de Classe: Calcula o acoplamento a classes exclusivas por meio de parâmetros, variáveis locais, tipos de retorno, chamadas de método, instanciações genéricas ou de modelo, classes base, implementações de interface, campos definidos em tipos externos e decoração de atributo.
- Linhas de código-fonte: Indica o número total de linhas no arquivo especificado, incluindo linhas em branco.
- Linhas de código executável: Calcula o número aproximado de linhas ou operações executáveis no código.

Tais métricas servem de parâmetro para determinar tanto o custo de implementação, quanto de manutenção e também performance. Podem servir de base para refatorações e otimizações no código.

### 4.6. Implementações adicionais

Além da demanda requerida pela BTG, foi desenvolvido também *endpoints* adicionais, que executavam a mesma funcionalidade que os anteriores, mas com diferenças em suas estratégias de implementação. Tais métodos foram desenvolvidos a fim de se estudar formas de otimizar o processamento necessário para entregar o resultado esperado.

#### 4.6.1. *Join* em memória

A primeira maneira testada consistiu em ao invés de utilizar o comando *JOIN* do *SQL*, requisitar todos os registros de cada tabela separadamente e então conectá-los usando a função *JOIN* do C#.

```
Field<ListGraphType<ContextoType>>("contextoInMem",
    arguments: new QueryArguments(
        new QueryArgument<StringGraphType> { Name = "filtro" }
    ), resolve: context =>
    {
        var filtro = context.GetArgument<string>("filtro");

        var registros1 = _dBConnection.GetContexto1();
        if (filtro == null)
        {
            return registros1;
        }

        var registros2 = _dBConnection.GetContexto2();

        var registros3 = _dBConnection.GetContexto3(filtro);

        var query = from registro1 in registros1
                    join registro2 in registros2 on registro1.ID equals registro2.ID_REGISTRO1
                    join registro3 in registros3 on registro1.ID_REGISTRO3 equals registro3.ID
                    select new
                    {
                        registro1 = registro1,
                        registro2 = registro2,
                        registro3 = registro3,
                    };

        return query;
    });
```

Figura 08 – Implementação da estratégia no *GraphQL*. Fonte: O Autor.

```
public List<Contexto> GetRegistrosContexto(string filtro = "")
{
    string queryOne = $"SELECT ta.* FROM CONTEXTO.TB_CONTEXTO1 as ta";
    string queryTwo = $"SELECT rca.* FROM CONTEXTO.TB_CONTEXTO2 as rca";
    string queryThree = $"SELECT tlp.* FROM CONTEXTO.TB_CONTEXTO3 as tlp WHERE contextoTres.TYPE = {filtro}";

    var resultOne = _dbContext.Database.SqlQuery<ContextoUm>(queryOne).ToList();
    var resultTwo = _dbContext.Database.SqlQuery<ContextoDois>(queryTwo).ToList();
    var resultThree = _dbContext.Database.SqlQuery<ContextoTres>(queryThree).ToList();

    var joinedResult = from contextoUm in resultOne
                join contextoDois in resultTwo on contextoUm.ID equals contextoDois.ID_CONTEXTO_UM
                join contextoTres in resultThree on contextoUm.ID_CONTEXTO_TRES equals contextoTres.ID
                select new
                {
                    contextoUm = contextoUm,
                    contextoDois = contextoDois,
                    contextoTres = contextoTres,
                };
    return joinedResult;
}
```

Figura 09 – Implementação da estratégia na *REST*. Fonte: O Autor.

### 4.6.2. Conexão com dois bancos de dados

Outro teste realizado foi a inclusão da conexão com dois bancos de dados diferentes. Apesar de não ser uma demanda pedida neste momento, a empresa citada anteriormente, utiliza em algumas ocasiões este tipo de situação.

```
[Route("Contexto/GetRegistrosContextoDoisDB")]
[HttpGet]
0 references | 0 changes | 0 authors, 0 changes
public HttpResponseMessage GetRegistrosContextoDoisDB([FromUri] string filtro)
{
    try
    {
        var response = _dBConnection.GetContextoRegistros(filtro);

        var dummyRequest = _anotherDBConnection.GetContextoRegistrosDois();

        return Request.CreateResponse(HttpStatusCode.OK, response);
    }
    catch (Exception ex)
    {
        return Request.CreateResponse(HttpStatusCode.InternalServerError, ex.Message + ex.InnerException);
    }
}
```

Figura 08 – Implementação da estratégia na *REST*. Fonte: O Autor.

```
Field<ListGraphType<ContextoType>>("contextoTwoDb",
    arguments: new QueryArguments(
        new QueryArgument<StringGraphType> { Name = "filtro" }
    ), resolve: context =>
    {
        var filtro = context.GetArgument<string>("filtro");

        if (filtro == null)
        {
            throw new Exception("No filter especified");
        }

        var registrosContexto = _dBConnection.GetRegistrosContexto(filtro);

        var registrosContextoDois = _anotherDBConnection.GetRegistrosContextoDois();

        return registrosContexto;
    });
```

Figura 09 – Implementação da estratégia no *GraphQL*. Fonte: O Autor.

## 5. Resultados e discussões

Após desenvolver todos os *endpoints*, utilizando as diferentes estratégias apresentadas, foram feitas 03 (três) rodadas de testes de performance em cada uma, utilizando o gatling. Os testes começavam fazendo uma requisição por segundo e aumentava esse número para cinco ao longo de dez segundos, quando finalizava a execução. Tais números foram decididos a partir de conhecimentos prévios do ambiente da BTG, tendo como noção que estes seriam números baixos, o que não faria com que o resultado fosse interferido por questões de alta carga de requisições. A seguir são exibidos os resultados dos testes com cada *endpoint*:

*Endpoint REST* com conexão a um banco de dados e utilizando o comando *JOIN* na query *SQL*:

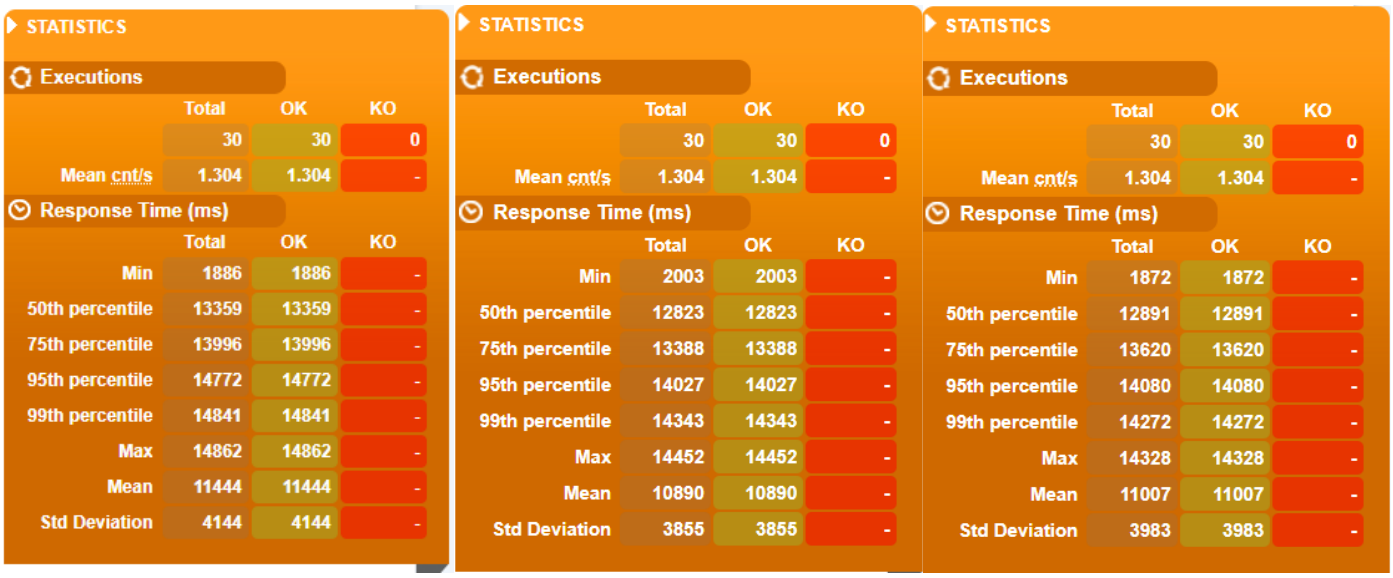


Figura 10 – Mosaico de imagens com os resultados da 1ª, 2ª e 3ª execuções do teste na aplicação *REST*, respectivamente. Fonte: O Autor.

*Endpoint REST* com conexão em dois bancos de dados e utilizando o comando *JOIN* no *SQL*:

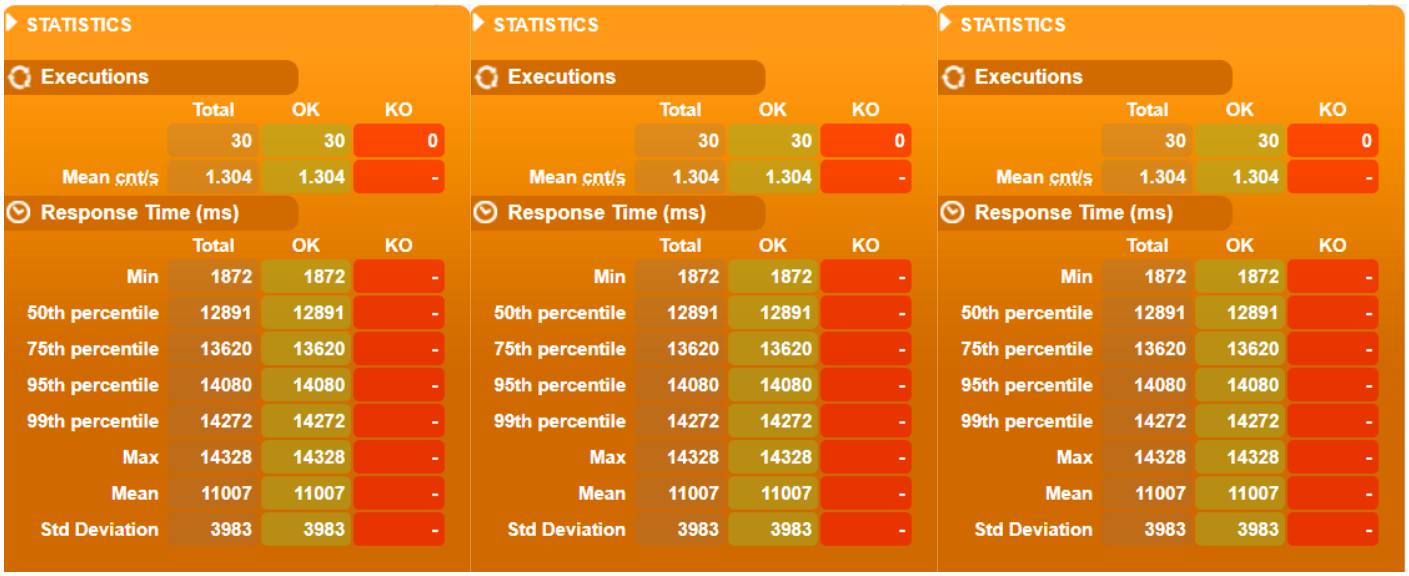


Figura 11 – Mosaico de imagens com os resultados da 1ª, 2ª e 3ª execuções do teste na aplicação *GraphQL*, respectivamente. Fonte: O Autor.

*Endpoint GraphQL* com conexão em um banco de dados, utilizando o comando *JOIN* no *SQL* e pedindo como retorno apenas um campo dos disponíveis:

STATISTICS

Executions

| | Total | OK | KO |
|---|---|---|---|
| | 30 | 30 | 0 |
| Mean cnt/s | 2.727 | 2.727 | - |

Response Time (ms)

| | Total | OK | KO |
|---|---|---|---|
| Min | 244 | 244 | - |
| 50th percentile | 253 | 253 | - |
| 75th percentile | 283 | 283 | - |
| 95th percentile | 1495 | 1495 | - |
| 99th percentile | 3081 | 3081 | - |
| Max | 3638 | 3638 | - |
| Mean | 476 | 476 | - |
| Std Deviation | 668 | 668 | - |

STATISTICS

Executions

| | Total | OK | KO |
|---|---|---|---|
| | 30 | 30 | 0 |
| Mean cnt/s | 2.727 | 2.727 | - |

Response Time (ms)

| | Total | OK | KO |
|---|---|---|---|
| Min | 240 | 240 | - |
| 50th percentile | 251 | 251 | - |
| 75th percentile | 260 | 260 | - |
| 95th percentile | 278 | 278 | - |
| 99th percentile | 299 | 299 | - |
| Max | 305 | 305 | - |
| Mean | 255 | 255 | - |
| Std Deviation | 13 | 13 | - |

STATISTICS

Executions

| | Total | OK | KO |
|---|---|---|---|
| | 30 | 30 | 0 |
| Mean cnt/s | 2.727 | 2.727 | - |

Response Time (ms)

| | Total | OK | KO |
|---|---|---|---|
| Min | 240 | 240 | - |
| 50th percentile | 253 | 253 | - |
| 75th percentile | 265 | 265 | - |
| 95th percentile | 347 | 347 | - |
| 99th percentile | 410 | 410 | - |
| Max | 429 | 429 | - |
| Mean | 266 | 266 | - |
| Std Deviation | 39 | 39 | - |

Figura 12 – Mosaico de imagens com os resultados da 1ª, 2ª e 3ª execuções do teste na aplicação *GraphQL*, respectivamente. Fonte: O Autor.

*Endpoint GraphQL* com conexão em dois bancos de dados, utilizando o comando *JOIN* no *SQL* e pedindo como retorno apenas um campo dos disponíveis**:**

STATISTICS

Executions

| | Total | OK | KO |
|---|---|---|---|
| | 30 | 30 | 0 |
| Mean cnt/s | 2.727 | 2.727 | - |

Response Time (ms)

| | Total | OK | KO |
|---|---|---|---|
| Min | 244 | 244 | - |
| 50th percentile | 250 | 250 | - |
| 75th percentile | 255 | 255 | - |
| 95th percentile | 270 | 270 | - |
| 99th percentile | 295 | 295 | - |
| Max | 305 | 305 | - |
| Mean | 253 | 253 | - |
| Std Deviation | 12 | 12 | - |

STATISTICS

Executions

| | Total | OK | KO |
|---|---|---|---|
| | 30 | 30 | 0 |
| Mean cnt/s | 2.727 | 2.727 | - |

Response Time (ms)

| | Total | OK | KO |
|---|---|---|---|
| Min | 239 | 239 | - |
| 50th percentile | 249 | 249 | - |
| 75th percentile | 256 | 256 | - |
| 95th percentile | 268 | 268 | - |
| 99th percentile | 273 | 273 | - |
| Max | 274 | 274 | - |
| Mean | 251 | 251 | - |
| Std Deviation | 9 | 9 | - |

STATISTICS

Executions

| | Total | OK | KO |
|---|---|---|---|
| | 30 | 30 | 0 |
| Mean cnt/s | 2.727 | 2.727 | - |

Response Time (ms)

| | Total | OK | KO |
|---|---|---|---|
| Min | 236 | 236 | - |
| 50th percentile | 248 | 248 | - |
| 75th percentile | 259 | 259 | - |
| 95th percentile | 304 | 304 | - |
| 99th percentile | 400 | 400 | - |
| Max | 431 | 431 | - |
| Mean | 258 | 258 | - |
| Std Deviation | 36 | 36 | - |

Figura 13 – Mosaico de imagens com os resultados da 1ª, 2ª e 3ª execuções do teste na aplicação *GraphQL*, respectivamente. Fonte: O Autor.

Na execução dos *Endpoints* tanto *GraphQL* quanto *REST* com conexão a um banco de dados e utilizando o comando *JOIN* em memória na aplicação o gatling não obteve resultados com a mensagem de resposta ao fazer requisições para esses *endpoints*, por a performance ter sido pífia, retornando um erro de timeout ao fazer as *requests*.

*Endpoint GraphQL* com conexão em um banco de dados, utilizando o comando *JOIN* no *SQL* e pedindo como retorno todos os campos disponíveis:

STATISTICS

Executions

| | Total | OK | KO |
|---|---|---|---|
| | 30 | 30 | 0 |
| Mean cnt/s | 2.727 | 2.727 | - |

Response Time (ms)

| | Total | OK | KO |
|---|---|---|---|
| Min | 240 | 240 | - |
| 50th percentile | 249 | 249 | - |
| 75th percentile | 253 | 253 | - |
| 95th percentile | 257 | 257 | - |
| 99th percentile | 262 | 262 | - |
| Max | 264 | 264 | - |
| Mean | 249 | 249 | - |
| Std Deviation | 5 | 5 | - |

STATISTICS

Executions

| | Total | OK | KO |
|---|---|---|---|
| | 30 | 30 | 0 |
| Mean cnt/s | 2.727 | 2.727 | - |

Response Time (ms)

| | Total | OK | KO |
|---|---|---|---|
| Min | 240 | 240 | - |
| 50th percentile | 252 | 252 | - |
| 75th percentile | 257 | 257 | - |
| 95th percentile | 274 | 274 | - |
| 99th percentile | 349 | 349 | - |
| Max | 376 | 376 | - |
| Mean | 256 | 256 | - |
| Std Deviation | 24 | 24 | - |

STATISTICS

Executions

| | Total | OK | KO |
|---|---|---|---|
| | 30 | 30 | 0 |
| Mean cnt/s | 2.727 | 2.727 | - |

Response Time (ms)

| | Total | OK | KO |
|---|---|---|---|
| Min | 240 | 240 | - |
| 50th percentile | 254 | 254 | - |
| 75th percentile | 261 | 261 | - |
| 95th percentile | 275 | 275 | - |
| 99th percentile | 281 | 281 | - |
| Max | 282 | 282 | - |
| Mean | 256 | 256 | - |
| Std Deviation | 10 | 10 | - |

Figura 14 – Mosaico de imagens com os resultados da 1ª, 2ª e 3ª execuções do teste na aplicação *GraphQL*, respectivamente.

Analisando os resultados obtidos é possível observar que os *endpoints* em *GraphQL* tiveram uma performance muito acima do *REST*, além de ter um desvio padrão baixíssimo, o que indica uma aplicação mais estável e menos propensa a sofrer com sobrecargas. Isso pode ser observado na tabela abaixo, que utilizou como base os dados da implementação de cada tecnologia onde foi utilizado *JOIN* no *SQL*, por ter sido a mais performática de ambos:

| | Média de Tempo de Response | Média de desvio padrão |
|---|---|---|
| *REST* | 11113ms | 3994ms |
| *GraphQL* | 332ms | 240ms |

Figura 15 – Tabela explicitando média de tempo de *response* e de desvio padrão dos *endpoints REST* e *GraphQL*.

Além disso é possível fazer uma comparação entre as requisições *GraphQL* requisitando apenas um parâmetro e as requisições que requisitaram todos, que no total são 34. Nessa análise, conclui-se que para o tempo de response do servidor, a quantidade de informações trafegada não faz diferença significativa, sendo essa característica um benefício apenas para o usuário final utilizar menos banda larga e para garantir flexibilidade de consulta aos dados para o programador front-end que for consumi-los. O resumo destas informações também pode ser visto na tabela abaixo:

| *GraphQL* | Média de Tempo de Response | Média de desvio padrão |
|---|---|---|
| *1 parâmetro* | 254ms | 33ms |
| *34 parâmetros* | 253ms | 13ms |

Figura 16 – Tabela explicitando média de tempo de *response* e de desvio padrão dos *endpoints GraphQL* retornando 1 parâmetro e retornando 34 parâmetros.

As análises de índice de manutenção, complexidade ciclomática, profundidade de herança, acoplamento de classes, linhas de código fonte e linhas de código executável demonstram que o *GraphQL* é uma tecnologia mais custosa para ser implementada quando comparada à *REST,* diferentemente do que é citado em Gleison Brito (2015), que afirma, através de um estudo empírico, que o *GraphQL* teria um tempo de implementação mais baixo.

Levando em consideração as métricas obtidas, é possível afirmar também que o código *GraphQL* é mais complexo, gerando um esforço maior para mantê-lo. As métricas estão apresentadas abaixo e foram transcritas para uma tabela pois o resultado original não ficava legível:

| Arquivo | ContextoRepository | IContextoRepository | Contexto (model) |
|---|---|---|---|
| Índice de manutenção | 66 | 87 | 100 |
| Complexidade Ciclomática | 3 | 1 | 70 |
| Profundidade de Herança | 5 | 0 | 1 |
| Acoplamento de Classe | 12 | 4 | 1 |
| Linhas de Código-Fonte | 34 | 4 | 38 |
| Linhas de Código Executável | 14 | 2 | 0 |

Figura 17 – Tabela com as métricas dos arquivos da aplicação *REST* gerados com o Visual Studio.

| Arquivo | ContextoMutation | ContextoQuery | ContextoSchema |
|---|---|---|---|
| Índice de manutenção | 73 | 66 | 83 |
| Complexidade Ciclomática | 1 | 2 | 1 |
| Profundidade de Herança | 6 | 6 | 3 |
| Acoplamento de Classe | 12 | 14 | 6 |
| Linhas de Código-Fonte | 21 | 56 | 8 |
| Linhas de Código Executável | 4 | 7 | 2 |

Figura 18 – Tabela com as métricas da primeira parte dos arquivos da aplicação *GraphQL* gerados com o Visual Studio.

| Arquivo | Contexto (model) | ContextoType | DbConnection | IDbConnection |
|---|---|---|---|---|
| Índice de manutenção | 100 | 36 | 100 | 67 |
| Complexidade Ciclomática | 70 | 1 | 1 | 12 |
| Profundidade de Herança | 1 | 5 | 0 | 1 |
| Acoplamento de Classe | 0 | 6 | 2 | 14 |
| Linhas de Código-Fonte | 40 | 43 | 4 | 116 |
| Linhas de Código Executável | 0 | 72 | 0 | 38 |

Figura 19 – Tabela com as métricas da segunda parte dos arquivos da aplicação *GraphQL* gerados com o Visual Studio.

Com os dois resultados destacados nas figuras 15 e 16, é possível concluir que o *GraphQL* possui uma melhor performance. Ao utilizar as métricas de código propostas para análise, a REST possui uma complexidade de desenvolvimento menor.

Assim, quando surgir um contexto em que há de ser feita a escolha de qual tecnologia deve-se utilizar, o ideal é levar em consideração os fatores do ambiente e ver qual delas melhor se alinhará aos mesmos.

O *GraphQL,* por exemplo, possui um tempo maior de implementação, mas isso pode ser relevado caso o contexto demande uma aplicação com alta performance. Já o *REST* pode ser preferido em times com muitos desenvolvedores, onde uma menor complexidade de código é desejável.

## 6. Trabalhos futuros

Em trabalhos futuros, seria interessante explorar outros tipos de testes utilizando estas mesmas tecnologias, como testes de carga, para se ter noção de qual delas consegue lidar melhor com alta demanda de requisições.

É plausível também fazer uma análise deste trabalho e do Gleison Brito (2015), ou deste com o do Armin Lawi *et. al.* (2021), já que os resultados desses trabalhos são divergentes em relação à comparação do tempo de implementação e performance do *REST* e do *GraphQL.*

Outro ponto interessante para ser explorado são as tecnologias gRPC e AMQP, que também possuem seus próprios pontos positivos, negativos e casos de uso.

*Por fim, é também válido* realizar outros estudos com REST e GraphQL, utilizando outros cenários que possam vir a ser implementados em casos reais.

## 7. Conclusões

Os aplicativos estão finalizados e guardados em repositórios privados da BTG. Foi gerado também um relatório que servirá de base para futuras tomadas de decisões. A demanda interna da empresa foi alcançada. Após utilizar o estudo para comparar as duas tecnologias, foi perceptível que o ponto positivo do *GraphQL* é ser mais performático, mostrando-se aproximadamente 33 vezes mais rápido, enquanto que o *REST* possui um

menor tempo de implementação e possivelmente de manutenção também, por ter um menor acoplamento de classes e menor herança. Dessa forma, na empresa, foi optado o uso do *GraphQL*, já que era necessário obter uma performance alta e levando em consideração também que, por ter uma baixa rotatividade de desenvolvedores e estratégias para passagem de conhecimento, como palestras, a complexidade de manter e até mesmo implementar novas aplicações *GraphQL*, se diluiria.

## Referências


Brito, G., & Valente, M. T. (2020). **"REST vs GraphQL: A controlled experiment"**. In 2020 IEEE international conference on software architecture (ICSA) (pp. 81-91). IEEE.

Mavroudeas, G., Baudart, G., Cha, A., Hirzel, M., Laredo, J. A., Magdon-Ismail, M., ... & Wittern, E. (2021, November). **"Learning GraphQL query cost".** In 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE) (pp. 1146-1150). IEEE.

Wittern, E., Cha, A., & Laredo, J. A. (2018). **"Generating graphql-wrappers for rest (-like) apis"**. In International Conference on Web Engineering (pp. 65-83). Springer, Cham.

Brito, G., Mombach, T., & Valente, M. T. (2019). **"Migrating to GraphQL: A practical assessment"**. In 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER) (pp. 140-150). IEEE.

Cha, A., Wittern, E., Baudart, G., Davis, J. C., Mandel, L., & Laredo, J. A. (2020). **"A principled approach to GraphQL query cost analysis"**. In Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (pp. 257-268).

Lawi, A., Panggabean, B. L., & Yoshida, T. (2021). **"Evaluating GraphQL and REST API Services Performance in a Massive and Intensive Accessible Information System"**. Computers, 10(11), 138.

Wittern, E., Cha, A., Davis, J. C., Baudart, G., & Mandel, L. (2019). **"An empirical study of GraphQL schemas"**. In International Conference on Service-Oriented Computing (pp. 3-19). Springer, Cham.

Li, L., Chou, W., Zhou, W., & Luo, M. (2016). **"Design patterns and extensibility of REST API for networking applications"**. IEEE Transactions on Network and Service Management, 13(1), 154-167.

Brabra, H., Mtibaa, A., Petrillo, F., Merle, P., Sliman, L., Moha, N., ... & Gargouri, F. (2019). **"On semantic detection of cloud API (anti) patterns. Information and Software Technology"**, 107, 65-82.

**"Calcular métricas de código - Visual Studio (Windows) "**. (2022). Docs.microsoft.com. https://docs.microsoft.com/pt-br/visualstudio/code-quality/code-metrics-values?view=vs-2022

**“GraphQL: A query language for APIs”**. (2012). Graphql.org. https://graphql.org/

**“Gatling Open-Source Load Testing – For DevOps and CI/CD”**. (n.d.). Gatling Open-Source Load Testing. https://gatling.io/

**“O que é a API RESTful?”**. s.d. AWS. https://aws.amazon.com/pt/what-is/restful-api/