GraphQL no Desenvolvimento de Webservices para Sistemas Mobile de Forma Minimizar Tempo de Resposta

Jonas Silva Gomes[1], Marco Antônio Pereira Araújo[1]

[1]Instituto Federal do Sudeste de Minas Gerais
Campus Juiz de Fora – MG – Brasil

jonas.gomes98@icloud.com, marco.araujo@ifsudestemg.edu.br

Abstract. *When it comes to mobile applications it is important to think about the service architecture given the processing, memory and bandwidth limitations of devices. This paper aims to analyze and verify if the architecture proposed by Facebook, GraphQL, presents superior performance than REST, in order to reduce the cost of bandwidth and optimize the response time. To perform this experiment, we developed two APIs and compared the performance of each one, which was collected the time in milliseconds. This paper presents a statistical analysis of the data obtained and discusses the trade-off of each service approach.*

Resumo. *Quando se trata de aplicações móveis é importante pensar na arquitetura de serviço tendo em vista as limitações de processamento, memória e banda dos dispositivos. O presente trabalho busca analisar e verificar se a arquitetura proposta pelo Facebook, o GraphQL, apresenta desempenho superior ao REST, de forma a reduzir o custo gasto de banda para receber dados pela rede e otimizar o tempo de resposta. Para a realização deste experimento, buscou-se desenvolver duas APIs e comparar o desempenho de cada uma, onde foi coletado o tempo em milissegundos. Com isso, é apresentada uma análise estatística dos dados obtidos e discutindo o trade-off de cada abordagem de serviço.*

Palavras-chave: engenharia de software, arquitetura de sistemas, análise desempenho.

1. Introdução

Nos dias de hoje, a sociedade globalizada movimenta uma grande quantidade de informações constantemente pela web, quando usuários acessam e alteram dados. Atualmente, a disponibilidade dos dados não é o único fator crucial para um webservice eficiente, mas também o tempo que demora para carregar esses dados. Outro fator relevante para construção de um webservice é a disponibilidade das informações através de um leque diferente de plataformas de rede, seja *4G*, *3G* ou *WiFi*.

A indústria de desenvolvimento de software desfruta de webservices para disponibilizar informações às quais usuários podem acessar por meio de diversas aplicações, seja web, móvel ou IoT. Quando se trata de aplicações móveis é importante pensar na arquitetura de serviço tendo em vista as limitações de processamento, memória e banda dos dispositivos. Segundo o SOA Manifesto criado em 2009, é possível utilizar arquiteturas como CORBA, SOAP, DCOM e REST. Porém com a evolução da tecnologia, objetos JSON apresentaram desempenho superior em relação aos envelopes

XML. Serviços do tipo RESTful são amplamente utilizados [Helgason 2017], pois esse tipo de serviço fornece a aplicação web, por meio de funções e recursos que acessam facilmente a base de dados entre o cliente e servidor, sem que o cliente interaja diretamente com o *backend*. Além das APIs (*Application Programming Interface*) REST, foi desenvolvido pelo Facebook uma nova arquitetura de integração chamada de GraphQL que foi definida como uma nova forma de acessar dados como uma tecnologia SOA. O GraphQL foi descrito como "uma linguagem de consulta para sua API" [Ghebremicael 2017], é considerado uma alternativa ao REST e foi construído em resposta a um problema de desempenho durante a mudança do Facebook para aplicativos móveis nativos.

No âmbito do projeto de ensino "Desenvolvimento de aplicativo para dispositivos móveis para disponibilização de oportunidade de emprego e estágio" do IF Sudeste MG no campus Juiz de Fora, observou-se que a plataforma teria muito potencial de uso, pois a divulgação de vagas de emprego e estágio atualmente é feita através dos professores em grupos do Facebook ou por e-mail. Ao adotar a plataforma possibilitaria melhorar a comunicação e atingir mais alunos, como os alunos recém-formados que buscam ingressar no mercado de trabalho e que deixam de receber esses e-mails. Dessa forma, construir uma aplicação escalável e otimizada seria extremamente necessário, uma vez que, além da instituição, empresas da região poderiam se interessar em divulgar suas oportunidades de emprego e estágio.

Com isso, o presente trabalho buscou analisar e verificar se a arquitetura proposta pelo Facebook, o GraphQL, apresenta melhores resultados em relação a arquitetura REST, de forma otimizar o tempo de resposta para que as empresas cadastrem suas vagas e os usuários acessem a essas vagas, podendo se candidatar a elas através de dispositivos móveis. E por fim fornecer um estudo científico para ajudar aos desenvolvedores na tomada de decisão entre qual arquitetura utilizar REST ou GraphQL.

1.1 Objetivos

O presente trabalho busca analisar e verificar se a arquitetura proposta pelo Facebook, o GraphQL, apresenta melhores resultados no desempenho rede, otimizar o tempo de resposta das requisições do servidor.

1.2 Objetivos Específicos

Os objetivos específicos desta pesquisa são: Investigar o estado da arte da engenharia de serviços para dispositivos móveis; fazer um experimento científico bem documentado, servindo como base para trabalhos futuros relacionados; promover uma discussão científica sobre os resultados obtidos no experimento; concluir qual arquitetura apresenta melhor desempenho no contexto do projeto.

2. Aportes Teóricos

Neste capítulo será abordada toda base teórica e tecnológica utilizada na pesquisa. As informações mais relevantes estarão nas sessões subsequentes para uma melhor compreensão do experimento conduzido. Grande parte das pesquisas acadêmicas tem seu ponto de partida pela revisão de literatura, porém, caso essa revisão não seja feita corretamente, não terá muito valor científico [Kitchenham 2004]. Esse é o principal motivo o qual se deve realizar uma boa revisão para localizar lacunas, identificar

oportunidades de pesquisa em trabalhos futuros, avaliar e interpretar a questão pertinente a uma pesquisa em particular [Kitchenham 2004]. Entretanto, há outras razões específicas que podem justificar o uso da revisão sistemática [Kitchenham 2004]:

- Resumir determinada evidência existente sobre alguma teoria ou tecnologia específica;
- Encontrar lacunas para a pesquisa em questão, possibilitando a definição de áreas as quais podem ser abertas mais investigações;
- Fornecer um embasamento teórico para novas pesquisas relacionadas.

O escopo de aplicação dessa revisão sistemática da literatura é relacionado com a utilização de arquiteturas de integração entre sistemas, focando a integração de plataformas distribuídas, mais especificamente, dispositivos móveis.

2.1 Revisão Sistemática da Literatura

De acordo com o Center for Evidence Based Management [Management 2019], uma das características principais de uma boa pesquisa baseada em evidencias, é a realização de questionamentos práticos bem construídos. O PICOC é conhecido como um método para descrever cinco itens de uma pergunta de pesquisa. "PICOC" é um acrônimo que significa: População, Intervenção, Comparação, Resultado e Contexto. Através do PICOC pode-se pensar sobre o tipo de pergunta que está fazendo e, portanto, que tipo de pesquisa forneceria a melhor resposta. A Tabela 1 descreve o PICOC.

Tabela 1. PICOC

PICOC	Palavras-Chave
População	Arquitetos de software; Engenheiros de software; Desenvolvedores;
Intervenção	REST, GRAPHQL
Comparação	REST x GRAPHQL em rede local e externa
Resultado	Reduzir tempo de resposta e tamanho de pacote de dados transferidos pela rede
Contexto	Dispositivos móveis

A partir do escopo deste trabalho, a questão da pesquisa foi definida: A arquitetura GraphQL é uma solução melhor para reduzir o tempo de resposta e tamanho de pacote de dados em dispositivos móveis, em relação ao REST?

2.2 Protocolo de Seleção de Estudos de Base

Para selecionar os estudos mais relevantes, foi aplicada uma estratégia de busca parar encontrar potenciais artigos (Tabela 2). Os artigos identificados foram escolhidos através da leitura e verificação dos seguintes critérios de inclusão e exclusão:

- **Avaliação da Qualidade dos Estudos Primários**: não foi definida uma metodologia para se avaliar a qualidade dos estudos, porém toda a abordagem qualitativa está baseada na fonte de extração do material e na aplicação dos critérios de inclusão e exclusão dos artigos;

- **Estratégia** de Extração de Informação: para cada artigo encontrado, foram extraídos os seguintes dados: Título do artigo; Resumo e Abstract; Contexto e tecnologia da aplicação; Descrição das metodologias utilizadas;
- **Sumarização** de Resultados: os resultados foram elencados e foram analisados para obter as pesquisas que comparam e realizam testes de desempenho nas arquiteturas em estudo;
- **Busca**: foi necessário restringir o universo das buscas, tal restrição pode ser feita através de uma *string* de busca, uma *string* de busca é utilizada para pesquisar em bibliotecas digitais, usando-se palavras chaves e analisando em todo o texto ou apenas no abstract a incidência dessas palavras. A *string* de busca utilizada para a questão da pesquisa foi: "(*SOA* OR "*Service Oriented Architecture*") AND *REST* AND *graphql* AND *service* AND *mobile* AND *performance*".

Tabela 2. Critérios de inclusão e exclusão

Critério	Descrição
Seleção de Fontes	Será fundamentada em bases de dados eletrônicas incluindo as conferências e artigos listados a seguir.
Palavras-chave	Arquiteturas de Integração, dispositivos móveis, desempenho, tempo de resposta, tamanho de pacote de dados.
Idioma dos Estudos	Português e Inglês.
Métodos de busca de fontes	As fontes serão acessadas via web. No contexto desta revisão não será considerada a busca manual.
Listagem de fontes	Google Acadêmico.
Tipo dos Artigos	Teórico, Prova de conceito, Estudos experimentais.
Critérios de Inclusão e Exclusão de artigos	Artigos que realizaram análise estatística do desempenho de ambas arquiteturas, GraphQL e REST.

Como resultado da busca realizada no Google Acadêmico, única máquina de busca com acesso disponível para o autor no momento da pesquisa, definido na lista de fontes dos critérios para realização da Revisão Sistemática da literatura, foram encontrados 83 resultados os quais, após aplicados os critérios de inclusão e exclusão, 10 artigos foram selecionados. Feito o levantamento dos estudos encontrados, para filtrar os artigos que possuíam maior grau de relevância para a pesquisa, foi obedecida a presença dos seguintes critérios:

- Apresentação clara de quais critérios seriam comparados entre as arquiteturas;
- Descrição do ambiente utilizado durante a execução dos experimentos;
- Possibilidade de usar o serviço por dispositivos móveis;
- Descrição de quais tipos de rede foram feitos os testes;
- Medição do tempo entre a requisição e a resposta do servidor;
- Padronização no retorno das requisições.

2.3 Trabalhos Relacionados

A área de integração de sistemas começou a ganhar destaque quando as organizações sentiram a real necessidade de comunicar seus sistemas entre si e até com sistemas externos. Com isso, a linha de pesquisa de engenharia de software para sistemas distribuídos vem crescendo ao longo dos anos e trabalhos que buscam analisar o desempenho das arquiteturas disponíveis evoluiu gradativamente.

A pesquisa de Belqasmi [Belqasmi et al. 2012] buscou analisar o desempenho de pacotes XML contra objetos JSON, traçando um comparativo entre a arquitetura SAOP e REST.

Maeda [Maeda 2012] pesquisou o desempenho da serialização e chega à conclusão de que a serialização binária pode ser muito mais eficaz que os formatos de serilização de texto como XML ou JSON.

Sumaray e Kami Makki Sumaray and Makki 2012] apresentam resultados da serialização binária e de base de texto em plataformas móveis e confirma que uma serialização binária é mais rápida também em plataformas móveis.

Villamiza [Villamizar et al. 2015] apresenta um estudo de caso em que o desempenho de diferentes arquiteturas de microsserviço é comparado com aplicativos monolíticos com a mesma funcionalidade.

Messina [Messina et al. 2016] apresentam uma revisão do padrão de design intimamente relacionado à arquitetura de microsserviço e fornece uma boa visão geral dos problemas de design enfrentados ao optar por uma arquitetura de microsserviço.

Tihomirovs e Grabis [Tihomirovs and Grabis 2016] apresentam uma análise de desempenho dos serviços Web baseados em REST e SOAP, nos quais o REST tem melhor desempenho.

Já Ghebremicael [Ghebremicael 2017] implementou uma ferramenta que buscava mesclar o melhor de cada uma de forma reduzir o tempo de resposta para as aplicações.

Soderlund [Soderlund 2017] verificou o desempenho da arquitetura REST em diversos frameworks escritos linguagens de programação diferentes.

Helgason [Helgason 2017] fez uma análise de performance aplicando o GraphQL e o REST visando descobrir qual apresentou melhor desempenho para requisições de consulta, e concluiu que o REST é mais rápido para as consultas estruturadas consideradas mais simples, como a busca de informações de apenas uma tabela ou fonte de dados. Por outro lado, se tratando de requisições que precisem acessar recursos vinculados a mais de duas tabelas, verificou-se que o GraphQL foi mais rápido.

Taskula [Taskula et al. 2019] fizeram estudos comparativos entre as vantagens e desvantagens do GraphQL em relação ao REST. Os métodos comparativos selecionados foram: análise de performance; analise qualitativa de complexidade; analise qualitativa de manutenibilidade. Esse trabalho focou em discutir o desempenho de requisições de leitura das arquiteturas, em redes Wi-Fi e 3G, que envolveram diversos recursos presentes em diferentes tabelas no banco de dados.

Os trabalhos relacionados trazem abordagens focadas no desempenho do lado do servidor e sistemas web. Esse trabalho buscou preencher lacunas deixadas na literatura, como a análise de desempenho das arquiteturas REST e GraphQL visando dispositivos

móveis *Android* e *iOS*, uma vez que pouco aborda-se tais dispositivos nos estudos citados. Seguindo na mesma linha de pesquisa de Helgason [Helgason 2017] e Taskula [Taskula et al. 2019] o presente artigo acrescenta à discussão um estudo focado em testes de desempenho entre as arquiteturas de forma sistemática, tanto para requisições de leitura quanto requisições de escrita para dispositivos móveis.

2.4 Referencial Teórico

A seguir será apresentado o referencial teórico sobre webservices, as arquiteturas em discussão, REST e GraphQL, e as tecnologias adotadas para realização do experimento, desde a linguagem de programação, tipo de banco de dados e frameworks utilizados.

2.4.1 Webservice

Quando o desenvolvimento Web revolucionou a forma como as organizações trocavam informação na década de 1990, os computadores dos clientes passaram a ter acesso a informações armazenadas em servidores remotos fora de suas próprias organizações. Porém, o acesso era exclusivamente por meio de um navegador Web e o acesso direto à informação por outros programas não era prático. Para melhorar a comunicação, foi proposta a ideia de *webservice* [Sommerville 2011]. Usando webservices, as empresas que desejam disponibilizar suas informações de forma global, poderiam fazê-lo definindo e publicando uma interface de webservice. Tal interface define quais os dados disponíveis e a forma com que eles podem ser acessados. Normalmente, o webservice permite consultar dados e até manipulá-los, como por exemplo, disponibilizar uma funcionalidade interna da organização para que um fornecedor atualize o estado de um pedido.

2.4.2 Microservice

Microsserviço ou m*icroservice,* é um estilo de arquitetura para sistemas de software [Johansson 2017] que foca na construção de software altamente escalável para a funcionar em nuvem. Os microsserviços devem ser pequenas partes independentes e acessíveis por outros sistemas independentes que, juntos, atendem ao propósito de um aplicativo ou sistema. Cada serviço presente nesta arquitetura é projetado e desenvolvido para lidar com um subdomínio ou capacidade de negócio e pode ser usado como parte de qualquer aplicativo, pois eles estão contidos em seu próprio processo [Johansson 2017].

Os microsserviços foram derivados da arquitetura orientada a serviços, exceto pelo fato de dever ser um webservice pequeno e lidar apenas com uma propriedade de negócio. Os microsserviços não estão vinculados a nenhuma estrutura, linguagem ou software específico e, o mais importante, não precisam ser construídos com a mesma tecnologia [Johansson 2017]. Para alcançar uma implementação paralela produtiva de serviços usando tecnologias diferentes, é fundamental existirem interfaces predefinidas que definam como a comunicação entre os serviços deve ocorrer. A comunicação interna pode ser síncrona ou assíncrona, ou seja, sequencial ou paralela, devendo sempre ser feita com chamadas de rede para garantir que os serviços sejam completamente desacoplados.

2.4.3 REST

Representational State Transfer [Belqasmi et al. 2012] ou simplesmente REST, é uma API do tipo cliente-servidor utilizada por aplicações que se comunicam via protocolo HTTP, tornando mais fácil e aceitável sua utilização, desde que o desenvolvedor não

precisa utilizar um protocolo particular. Os pilares do REST são endereçamento, interface uniforme e gerenciamento de estados [Belqasmi et al. 2012]. A ideia principal do REST é a combinação de *URIs* que formam um *endpoint* de API para acessar dados.

A API funciona baseada nos princípios de CRUD (Criação, Leitura, Edição, Remoção), e utiliza métodos HTTP. O webservice construído com essa arquitetura é chamado de *RESTful* desde que respeite os padrões do HTTP. O REST possui muitos pontos fortes, tais como modularizar o sistema e delegar a responsabilidade de tratamento dos dados para cada método de um recurso específico [Taskula et al. 2019]. Não há necessidade de bibliotecas e ferramentas personalizadas para estabelecer uma conexão entre um cliente e um servidor e trocar dados entre eles. Além disso, o REST simplifica o design da interface da API de maneira legível por seres humanos, pois, afinal, tudo consiste em *URIs* simples e um conjunto limitado de verbos HTTP [Belqasmi et al. 2012].

2.4.4 GraphQL

O GraphQL é uma linguagem de consulta para APIs em tempo de execução desenvolvida pelo Facebook [Helgason 2017]. Essa tecnologia executa consultas no lado do servidor e retorna apenas os dados que são definidos por um sistema de tipos, dessa forma facilita a evolução das APIs ao longo do tempo. O novo paradigma gira em torno do cliente, pois são eles que controlam os dados que obtêm, não o servidor. As consultas do GraphQL acessam não apenas as propriedades de um recurso, mas também seguem as referências entre eles [Helgason 2017]. Embora as APIs REST exijam o carregamento de vários *endpoints*, as APIs GraphQL obtêm todos os dados que o aplicativo cliente precisa em uma única requisição, ou seja, consultas específicas podem ser construídas com base nos serviços GraphQL que pré-definiram os dados disponíveis para tal, permitindo um único *endpoint* em vez de múltiplos [Taskula et al. 2019]. As APIs que utilizam dessa arquitetura são organizadas em termos de tipos e campos chamados de *schema*. Ao declarar campos, pode-se buscar, por exemplo, apenas os dados relevantes de uma vaga de trabalho de uma empresa, em vez de receber todos os detalhes relacionados a vaga, classificar e exibir apenas os campos necessários.

O GraphQL usa tipos para garantir que os aplicativos solicitem apenas o que é possível e forneçam erros claros. Ao eliminar o número de consultas e a quantidade de informações transferidos, a velocidade de transferência dos dados poderia ser melhorada, além de resolver problemas que a arquitetura REST apresenta, como a busca excessiva e a insuficiência de dados [Taskula et al. 2019]. O GraphQL no lado do servidor precisa incluir o que foi mencionado antes, *schema*. Esses *schemas* são modelos e podem ser traduzidos para as respostas da consulta. No caso desta pesquisa, os *schemas* estão lá para representar as coleções NOSQL e os campos que elas contêm. Um objeto GraphQL é criado para cada representação de coleção, contendo os campos dos objetos reais.

2.4.5 Tecnologias para Desenvolvimento para Smartphones

Há diversas tecnologias para o desenvolvimento de aplicativos para dispositivos móveis atualmente. Podem ser divididas em abordagens nativas, abordagens híbridas e abordagens web. Analisou-se qual a tecnologia existente que atendesse às exigências do projeto: desenvolver para dois sistemas operacionais visando contemplar maior casos de testes possíveis. As opções encontradas foram: Java para o sistema *Android*, *Swift* para o *IOS*, ambas nativas. O *Ionic Framework* para plataformas web. O *Xamarin* e *React Native* para multiplataformas.

A desvantagem de se construir aplicativos nativos é demandar conhecimento específico em cada linguagem de sistema operacional. Essa desvantagem foi o principal fator para a não utilização dessa abordagem no contexto deste trabalho. As aplicações móveis nativas possuem ainda custos financeiros mais altos e precisam seguir à risca as políticas de desenvolvimento na qual estarão disponíveis [DiMarzio 2016]. As aplicações web são muito comuns entre desenvolvedores. Apresentam funcionalidades básicas e são multiplataformas, tendo como pré-requisito apenas um *browser* e conexão com a internet. Aplicações híbridas são conhecidas por possuírem recursos nativos e tecnologias web. Dessa forma, a ferramenta desenvolvida pode ser disponibilizada nas lojas virtuais e possui acesso às funcionalidades do s*martphone*. Isso é possível pois, apesar de possuírem uma linguagem originada na web, o aplicativo resultante é modelado com codificação nativa. Sua usabilidade é similar ao aplicativo nativo e essa é uma opção que permite estar presente em várias lojas. As aplicações de multiplataforma consistem em utilizar uma única linguagem para gerar o mesmo produto em plataformas diferentes e com desempenho semelhante, senão igual, as aplicações nativas. As principais são *Xamarin* e o *React-Native*. O *Xamarin* permite criar aplicativos usando o *C#* como linguagem de desenvolvimento. Possui uma ponte para as bibliotecas nativas de cada plataforma, com isso o desenvolvedor tem acesso a todas as APIs existentes no Sistema Operacional. O *React Native* é uma ferramenta que utiliza *JavaScript* como linguagem de programação e utiliza o React como *framework* para desenhar a interface de usuário do seu aplicativo [Eisenman 2017]. No *React Native*, não é gerado um código nativo, seu núcleo serve como uma ponte entre o que é nativo e o *JavaScript* da sua aplicação [Eisenman 2017].

3. Metodologia

A arquitetura de serviços utilizada no desenvolvimento de software é indispensável para sistemas distribuídos atualmente. O profissional responsável por realizar essa implementação precisa conhecer as melhores opções de integração de sistemas, a fim de fornecer o serviço mais rápido e otimizado aos clientes que utilizarem plataformas móveis. A pesquisa tem como objetivo analisar e comparar quais as abordagens mais utilizadas no desenvolvimento de softwares móveis. Uma vez selecionada a tecnologia, foi necessário fazer um estudo em cima das bibliotecas disponíveis que pudessem acelerar o desenvolvimento da ferramenta proposta.

O Node.js é um ambiente de tempo de execução que pode executar código *JavaScript* independente do navegador, ele foi projetado para fornecer ao programador uma maneira de construir aplicativos escaláveis e apresenta um desempenho mais rápido por utilizar do mecanismo *JavaScript V8* do *Chrome* para converter o código *Javascript* em código de máquina de baixo nível [Moilanen et al. 2019]. Pode-se usar o Node.js para criar serviços da Web, por desenvolvedores que preferem usar o *JavaScript* como linguagem principal, dessa forma os aplicativos *frontend* e a lógica do *backend* são escritos em uma linguagem comum. O Node.js utiliza o NPM (*Node Package Manager*) para gerenciar facilmente as dependências do projeto [Moilanen et al. 2019]. O JavaScript usa o JSON como formato de notação de objeto nativo, o que significa que o cliente, o servidor e o banco de dados podem usar o mesmo formato. Além disso, o Node.js possui uma E/S sem bloqueio e orientada a eventos [Moilanen et al. 2019].

Existem muitas estruturas do Node.js disponíveis para uso. O Express é um framework de roteamento minimalista que cria um servidor HTTP sobre o Node.js, é considerado fácil de usar, escalável e flexível para vários propósitos [Moilanen et al.

2019]. Além disso, há uma infinidade de soluções prontas disponíveis devido à grande base de usuários. Para a implementação das APIs utilizou-se esse framework, por ser uma biblioteca para abstrair a escrita de APIs, uma vez que possui diversos métodos e *middlewares* construidos em Node.js [Moilanen et al. 2019].

Os bancos de dados não relacionais, também conhecidos como bancos de dados NoSQL, diferem bastante dos bancos relacionais [Moilanen et al. 2019]. Geralmente, os bancos de dados NoSQL são especializados em vários casos de uso, pois foram projetados para superar dificuldades técnicas específicas de SGBD (Sistemas Gerenciadores de Banco de Dados) tradicionais. Embora situacionais, os bancos de dados NoSQL são de vários tipos, incluindo Chave-Valor, Coluna Ampla, Repositório de Documentos e Armazenamento em Grafos [Carpilovsky 2019]. As arquiteturas de Chave-Valor guardam os dados como sequências de caracteres, números ou datas que são mapeados por uma chave. Os Repositórios de Documentos também são uma espécie de armazenamento do tipo Chave-Valor, a principal diferença é que o valor é armazenado como um objeto no formato JSON, proporcionando muito mais complexidade e flexibilidade [Moilanen et al. 2019]. O armazenamento de Colunas Amplas é semelhante aos bancos de dados relacionais, pois consistem em tabelas com linhas e colunas. Já os bancos de dados em Grafo são coleções de nós interconectados, em que cada nó é um objeto com propriedades. Os nós são conectados unidirecionalmente a outros nós de maneira relacional [Moilanen et al. 2019].

Nesta pesquisa, foi selecionado o Node.js para escrever ambas APIs em REST e GraphQL devido a sua compatibilidade com o *JavaScript*. Embora o próprio GraphQL possa ser usado com muitas outras linguagens de programação, o Node.js foi escolhido pois, dessa forma, o ambiente de teste será simplificado. Através da medição do tempo de resposta, foi realizado um comparativo sobre qual arquitetura se mostrou mais eficiente. Em relação aos dispositivos móveis, foi utilizado o *React Native* para criar as aplicações móveis gerando dois aplicativos, um para Android e outro para iOS. Antes de realizar os testes definitivos, foi feito um pré-teste em rede local para averiguar o funcionamento dos serviços desenvolvidos. O experimento final foi dividido em duas fases principais. Requisições do tipo leitura e requisições do tipo inserção de dados. Para a API REST, as requisições de leitura são feitas através do protocolo HTTP usando o método GET. Já para API GraphQL, todas as requisições HTTP são do tipo POST, porém as operações de leitura são chamadas de *querys* e as de inserção, edição ou remoção são chamadas de *mutations*. O ambiente técnico utilizado para o experimento conta com um dispositivo Android, um iPhone e um servidor em nuvem fornecido pela AWS, como consta na Tabela 3. Os dispositivos móveis fizeram todas as requisições com aplicativos comumente rodando em segundo plano, como redes sociais, mapas, browsers, entre outras aplicações de usuário, pois neste estudo, o foco está em traçar um comparativo entre arquiteturas com seu desempenho em ambientes reais.

Tabela 3. Dispositivos Utilizados

Dispositivo	Sistema	Fabricante	RAM	Processador	Rede
Redmi 7	Android	Xiaomi	3GB	Octa-core	Wifi
iPhone 7	iOS	Apple	2GB	Quad-core	Wifi
Servidor	Linux	AWS	4GB	Dual-core	Wifi

A Tabela 4 descreve os tamanhos dos pacotes utilizados nas requisições de escrita por cada arquitetura e a Tabela 5 descreve os tamanhos dos pacotes nas requisições de leitura.

Tabela 4. Pacote de dados requisição de escrita

Requisições de Escrita	REST	GRAPHQL
Cadastrar Empresa	338	60
Cadastrar Usuário Empresa	399	242
Cadastrar Usuário Cliente	462	242

Tabela 5. Pacote de dados requisição de leitura

Requisições de Leitura	REST	GRAPHQL
Login Cliente	1302	827
Listar Vaga por ID	1294	383
Listar 15 Vagas	19436	3890

4. Análise Estatística dos Dados

A análise de dados foi separada em duas sessões, requisições de escrita e leitura, em um universo de 300 amostras por tipo de requisição, de forma a facilitar o raciocínio e entendimento dos dados. Inicialmente foi investigado o tempo médio de resposta para as requisições de escrita e a análise do custo de banda da requisição em função do tamanho do pacote de dados recebido. Depois foi feita a mesma análise para requisições do tipo leitura. É importante ressaltar que o tamanho de requisição para as plataformas *Android* e *iOS* se manteve constante, entretanto o tempo de resposta apresentou variação entre os dispositivos. Após a coleta dos dados do experimento, utilizou-se a estatística descritiva para especificar características relevantes dos mesmos, através da ferramenta MiniTab versão 17, como pode ser visto na Figura 1.

Descriptive Statistics: Tempo

Variable	Arquitetura	N	N*	Mean	SE Mean	StDev	Minimum	Q1	Median	Q3
Tempo	GRAPHQL	300	0	616,9	14,4	249,8	252,0	280,0	746,5	805,8
	REST	300	0	364,44	3,93	68,04	257,00	302,00	380,00	391,75

Variable	Arquitetura	Maximum
Tempo	GRAPHQL	1549,0
	REST	849,00

Figura 1: Estatística Descritiva – Requisições de Escrita Android

Inicia-se a análise estatística dos dados pela verificação de normalidade dos dados, para a qual serão consideradas as colunas “Arquitetura” e “Tempo”. Para a realização desse teste no Minitab, deve-se selecionar no menu a opção “Stat > Basic Statistics > Normality Test ...” e para variável “Tempo”. O teste de normalidade deve ser aplicado para cada variável individualmente. Deve-se considerar as seguintes hipóteses, a um nível de significância de 5%:

• H0 (hipótese nula): As amostras apresentam distribuição normal.

• H1 (hipótese alternativa): As amostras não apresentam distribuição normal.

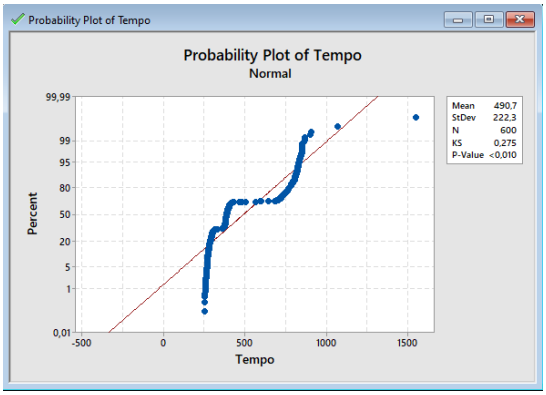

Figura 2. Análise de normalidade para a variável Tempo

Como pode ser observado nas Figura 2, as amostras não apresentam distribuição normal, com um *p-value* < 0,010 para a variável “Tempo”. Haja visto que uma das variáveis apresentam *p-value* menor que o nível de significância estabelecido de 5%, indicando a necessidade de um teste não paramétrico. Para a utilização de um método não paramétrico para o fator “Tempo”, será utilizado o método de Mann-Whitney. Para realização desse teste utilizou-se a comparação do “Tempo” entre cada arquitetura, REST e GraphQL. Esse teste é aplicado considerando as seguintes hipóteses, a um nível de significância de 5%:

• H0 (hipótese nula): Não há diferença entre as médias.

• H1 (hipótese alternativa): Há diferenças entre as médias.

O teste não paramétrico de Mann-Whitney pode ser realizado no Minitab por meio do menu “Stat > Nonparametrics > Mann-Whitney ...”. Conforme Figura 3, a qual exibe o teste de Mann-Whitney para verificação das médias na plataforma *Android*, constata-se que as amostras não possuem um nível de significância do ponto de vista estatístico, visto que apresentam um índice menor que 5%, logo aceita-se a hipótese alternativa de que há diferença significativa entre as médias. Conclui-se então que o REST teve um desempenho superior ao GraphQL em requisições de escrita, uma vez que teve um tempo médio menor.

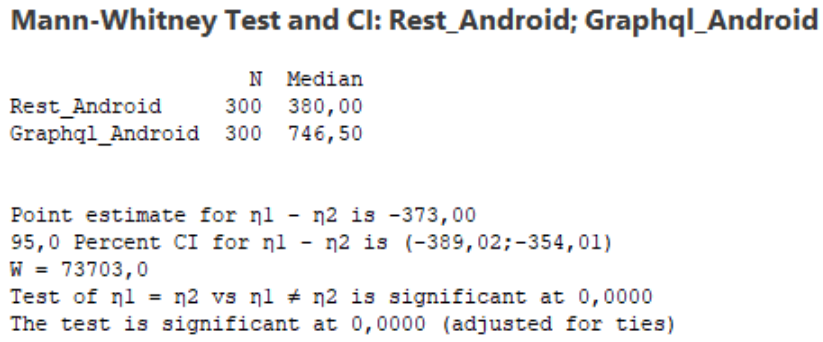

Figura 3: Mann-Whitney Test and CI Requisições de Escrita Android

Como a natureza dos dados foram as mesmas, mudando apenas a plataforma *Android* para *iOS*, os mesmos procedimentos foram adotados como a verificação de normalidade dos dados, para a qual foram consideradas as mesmas colunas "Arquitetura" e "Tempo". As amostras não apresentam distribuição normal, com um *p-value* < 0,010 para a variável "Tempo". Uma vez que o uma das variáveis apresentam *p-value* menor que o nível de significância estabelecido de 5%, utilizou-se do método não paramétrico de Mann-Whitney. Para realização desse teste utilizou-se a comparação do "Tempo" entre cada arquitetura, REST e GraphQL:

- H0 (hipótese nula): Não há diferença entre as médias.
- H1 (hipótese alternativa): Há diferenças entre as médias.

De acordo com a Figura 4, a qual exibe o teste de Mann-Whitney para verificação das médias na plataforma *iOS*, constata-se que as amostras não possuem um nível de significância do ponto de vista estatístico, visto que apresentam um índice menor que 5%, logo aceita-se a hipótese alternativa de que há diferença significativa entre as médias. Conclui-se então que o REST também obteve um desempenho melhor em relação ao GraphQL em requisições de escrita na plataforma *iOS*, uma vez que teve um tempo médio menor.

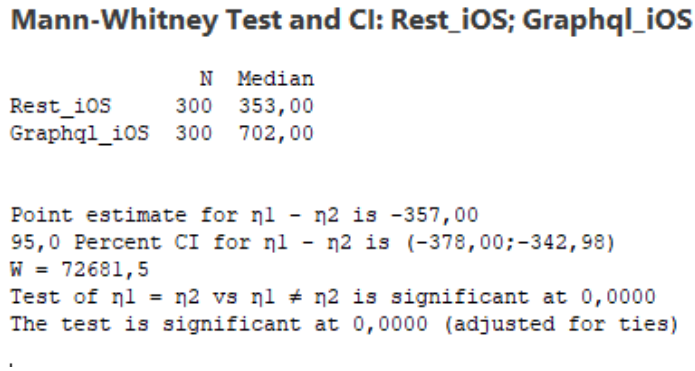

Figura 4: Mann-Whitney Test and CI Requisições de Escrita iOS

Uma vez realizada a análise estatística nas requisições de escrita, repetiu-se o procedimento como um novo grupo de amostras referentes as requisições de leitura, buscando identificar se o desempenho do REST se manteve superior ao GraphQL em

ambas plataformas, *Android* e *iOS*. A Figura 5 faz analise descritiva dos dados em das requisições de leitura no Android em relação ao tempo.

Descriptive Statistics: Tempo

Variable	Arquitetura	N	N*	Mean	SE Mean	StDev	Minimum	Q1	Median	Q3	Maximum
Tempo	GRAPHQL	300	0	523,5	13,9	240,3	313,0	343,3	373,0	773,5	2136,0
	REST	300	0	439,9	14,7	254,8	219,0	253,5	272,5	716,0	2049,0

Figura 5. Estatística Descritiva - Requisições de Leitura Android

Seguindo a metodologia adotada de análise, aplicou-se novamente o método de Mann-Whitney. Conforme Figura 6, a qual exibe o teste de Mann-Whitney para verificação das médias na plataforma *Android*, e a Figura 7 a qual demonstra o resultado do teste de Mann-Whitney para verificação das médias na plataforma *iOS*, constata-se que as amostras não possuem um nível de significância do ponto de vista estatístico, visto que apresentam um índice menor que 5%. Conclui-se então que o REST manteve o um desempenho melhor em relação ao GraphQL em requisições de leitura nas duas plataformas, tendo em vista que o tempo médio menor.

Mann-Whitney Test and CI: Rest_Android; Graphql_Android

	N	Median
Rest_Android	300	272,50
Graphql_Android	300	373,00

Point estimate for η1 - η2 is -91,00
95,0 Percent CI for η1 - η2 is (-98,99;-81,99)
W = 70144,5
Test of η1 = η2 vs η1 ≠ η2 is significant at 0,0000
The test is significant at 0,0000 (adjusted for ties)

Figura 6. Mann-Whitney Test and CI Requisições de Leitura Android

Mann-Whitney Test and CI: Rest_iOS; Graphql_iOS

	N	Median
Rest_iOS	300	301,00
Graphql_iOS	300	423,00

Point estimate for η1 - η2 is -98,00
95,0 Percent CI for η1 - η2 is (-113,00;-84,02)
W = 70666,5
Test of η1 = η2 vs η1 ≠ η2 is significant at 0,0000
The test is significant at 0,0000 (adjusted for ties)

Figura 7. Mann-Whitney Test and CI Requisições de Leitura iOS

5. Resultados e Discussão

Este estudo buscou demonstrar de forma consistente o desempenho das duas arquiteturas utilizadas para integração de sistemas no ambiente de dispositivos móveis. Uma vez coletado todos os dados referentes ao experimento, foi feito o teste de hipótese buscando responder à questão da pesquisa: Há melhora no desempenho da arquitetura GraphQL em relação ao tempo de resposta do servidor e a redução do tamanho de dados? Após a condução do experimento e análise dos dados, foi verificado que o desempenho médio para o tempo de resposta é pior, mas o desempenho no tamanho dos pacotes de dados é melhor que do REST (H2). Como isso, foi possível observar que o tempo de resposta é mais rápido em plataforma *iOS* do que na plataforma *Android* e que para redes locais a média se manteve constante como para redes externas.

Segundo Taskula [Taskula et al. 2019], um dos motivos para desencorajar utilização do REST é que um cliente precisa estar em conformidade com a interface uniforme fornecida pela API. Normalmente, a informação necessária é transferida de forma padronizada, possivelmente incluindo apenas o identificador de qualquer sub recurso ou todo o recurso solicitado, mesmo que apenas uma pequena parte desse recurso seja necessária ao aplicativo do cliente. Taskula desenvolveu um estudo que considera uma exibição de uma lista de todas as receitas pertencentes a uma padaria, na qual apenas os nomes das receitas são mostrados na lista. Ele chegou à conclusão de que, com a arquitetura REST, solicitar essas receitas envolve buscar todo o recurso de cada receita para listar apenas seus nomes. Isso é altamente ineficiente, especialmente para um grande número de recursos solicitados, e pode criar um gargalo dentro do aplicativo, degradando a experiência do usuário. Esse tipo de comportamento é chamado de busca excessiva de dados [Taskula et al. 2019], o qual a maioria dos dados solicitados não é realmente necessário para o cliente. Outra falha comum das APIs REST é a busca insuficiente de dados [Taskula et al. 2019], na qual o cliente precisa primeiro solicitar algum recurso e, em seguida, ler os identificadores de qualquer sub recurso e solicitá-los um por um para preencher os dados ausentes do recurso original. Dessa forma, como o REST trata-se de uma arquitetura centralizada em recursos, leva mais facilmente à busca excessiva e a insuficiência de dados do que a uma abordagem mais centrada em dados, como o GraphQL, que fornece apenas os dados solicitados pelo cliente.

6. Conclusão

No âmbito dos dispositivos móveis é importante pensar na arquitetura de serviço tendo em vista as limitações de processamento, memória e banda dos dispositivos. Com essa pesquisa, traçamos um comparativo entre duas arquiteturas de integração de sistemas webservice, dessa forma desenvolvedores durante o planejamento de software podem optar por qual arquitetura será mais interessante, analisando o *trade-off* de cada solução.

O REST é facilmente adotado pelos desenvolvedores pela facilidade de implementação, tanto do lado do cliente quanto do servidor, a modularização do sistema e o fato de utilizar protocolos simples, como HTTP e *URI*, tornam o REST amplamente difundido na implementação de webservice. O GraphQL, por outro lado, nasceu de um problema específico de desempenho do Facebook para dispositivos móveis e apesar de ser simples de implementar como o REST, apresentou desempenho pior que em relação ao tempo de resposta do servidor, entretanto, o GraphQL se mostrou excelente para a

redução do tamanho dos pacotes trafegados pela rede, deixando assim, o aplicativo mais leve e ágil em termos de processamento, tanto para o *Android* quanto para o *iOS*.

Portanto, o REST, por ter apresentado melhores resultados para o tempo de resposta em relação ao GraphQL, foi considerado como a melhor abordagem para o projeto de ensino "Desenvolvimento de aplicativo para dispositivos móveis para disponibilização de oportunidade de emprego e estágio" do IF Sudeste MG no campus Juiz de Fora. A pesquisa auxiliou no desenvolvimento do projeto, pois com os resultados estatísticos obtidos, pode-se escolher a melhor arquitetura para implementar, oferecendo um produto final mais rápido e eficiente. Além disso, observou-se que existem diferentes tecnologias para desenvolvimento mobile, sendo o *React-Native* mais recomendado pela comunidade *Open Source* mantida pelo Facebook, que deu suporte ao desenvolvimento da ferramenta "*Estag*".

Como indicação para trabalhos futuros, pode-se discutir a relação de desempenho dos dispositivos, no que diz respeito a economia de bateria. Também pode-se realizar o teste de desempenho em redes móveis *4G*, a qual este trabalho não contempla. Em relação às arquiteturas, pode-se desenvolver um estudo aprofundado no sistema de cache do REST visando a redução do tamanho do pacote de dados retornado.

Referências

Belqasmi, F., Singh, J., Melhem, S. Y. B., and Glitho, R. H. (2012). Soap-based vs. restful web services: A case study for multimedia conferencing. IEEE internet computing, 16(4):54–63.

Carpilovsky, A. (2019). APLICACAO WEB DE ANOTACOES PARA AUXILIO NA MEMORIZA-´ CAO DE VOCABULARIOS. PhD thesis, Universidade Federal do Rio de Janeiro.

CEDERLUND, M. (2016). Performance of frameworks for declarative data fetching.

DiMarzio, J. (2016). Beginning Android Programming with Android Studio, volume 4. Packt Publishing.

Eisenman, B. (2017). Learning React Native: Building Native Mobile Apps with JavaScript, volume 2. O'Reilly Media.

Eizinger, T. (2017). Api design in distributed systems: A comparison between graphql and rest.

Ghebremicael, E. S. (2017). Transformation of rest api to graphql for opentosca. Master's thesis.

Helgason, A. F. (2017). Performance analysis of web services: Comparison between restful & graphql web services.

Johansson, P. (2017). Efficient communication with microservices.

Kitchenham, B. (2004). Procedures for performing systematic reviews. Keele, UK, Keele University, 33(2004):1–26.

Maeda, K. (2012). Performance evaluation of object serialization libraries in xml, json and binary formats. In 2012 Second International Conference on Digital Information

and Communication Technology and it's Applications (DICTAP), pages 177–182. IEEE.

Management, C. E. B. (2019). What is a picoc? [Online; accessed 01-Setember-2019].

Messina, A., Rizzo, R., Storniolo, P., Tripiciano, M., and Urso, A. (2016). The database-isthe-service pattern for microservice architectures. In International Conference on Information Technology in Bio-and Medical Informatics, pages 223–233. Springer.

Moilanen, M. et al. (2019). Developing a web service: Databases, security and access control.

Soderlund, S. (2017). Performance of rest applications: Performance of rest applications in four¨ different frameworks.

Sommerville, I. (2011). Software engineering. Addison-Wesley/Pearson.

Sumaray, A. and Makki, S. K. (2012). A comparison of data serialization formats for optimal efficiency on a mobile platform. In Proceedings of the 6th international conference on ubiquitous information management and communication, page 48. ACM.

Sungkur, R. and Daiboo, S. (2015). Sorest, a novel framework combining soap and rest for implementing web services. In Proceedings of the Second International Conference on Data Mining, Internet Computing, and Big Data, pages 22–34.

Taskula, T. et al. (2019). Advanced data fetching with graphql: Case bakery service.

Tihomirovs, J. and Grabis, J. (2016). Comparison of soap and rest-based web services using software evaluation metrics. Information Technology and Management Science, 19(1):92–97.

Tran, T. (2019). Build a graphql application with node. js and react.

Villamizar, M., Garces, O., Castro, H., Verano, M., Salamanca, L., Casallas, R., and Gil, S. (2015). Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In 2015 10th Computing Colombian Conference (10CCC), pages 583–590. IEEE.