Marlon Diego Casagranda França

Desempenho na comunicação de dados entre cliente WEB e API WEB

Araquari – SC

Julho de 2017

Marlon Diego Casagranda França

Desempenho na comunicação de dados entre cliente WEB e API WEB

Trabalho de conclusão de curso apresentado como requisito parcial para a obtenção do grau de bacharel em Sistemas de Informação do Instituto Federal Catarinense.

Instituto Federal Catarinense – IFC

Câmpus Araquari

Bacharelado em Sistemas de Informação

Orientador: Prof. Dr. Eduardo da Silva

Coorientador: Prof. Dr. Eduardo da Silva

Araquari – SC

Julho de 2017

Marlon Diego Casagranda França
Desempenho na comunicação de dados
entre cliente WEB e API WEB/ Marlon Diego Casagranda França. – Araquari – SC, Julho de 2017-
?? p. : il. (algumas color.) ; 30 cm.

Orientador: Prof. Dr. Eduardo da Silva

Monografia (Graduação) – Instituto Federal Catarinense – IFC
Câmpus Araquari
Bacharelado em Sistemas de Informação, Julho de 2017.

1. Palavra-chave1. 2. Palavra-chave2. 2. Palavra-chave3. I. Orientador. II. Instituto Federal Catarinense. III. Câmpus Araquari. IV. Título

Marlon Diego Casagranda França

Desempenho na comunicação de dados entre cliente WEB e API WEB

Trabalho de conclusão de curso apresentado como requisito parcial para a obtenção do grau de bacharel em Sistemas de Informação do Instituto Federal Catarinense.

Trabalho aprovado. Araquari – SC, 24 de novembro de 2016:

Prof. Dr. Eduardo da Silva
Orientador

Professor
Convidado 1

Professor
Convidado 2

Araquari – SC
Julho de 2017

Dedicatória do trabalho de conclusão que deve ser algo breve e conciso.

Agradecimentos

Página com os agradecimentos do autor à pessoas importantes para a realização do trabalho.

"Colocar alguma frase importante que,
tenha motivado o autor no decorrer do desenvolvimento
do trabalho."
(Referência de local da frase)

Resumo

Segundo a **??**, 3.1-3.2), o resumo deve ressaltar o objetivo, o método, os resultados e as conclusões do documento. A ordem e a extensão destes itens dependem do tipo de resumo (informativo ou indicativo) e do tratamento que cada item recebe no documento original. O resumo deve ser precedido da referência do documento, com exceção do resumo inserido no próprio documento. (. . .) As palavras-chave devem figurar logo abaixo do resumo, antecedidas da expressão Palavras-chave:, separadas entre si por ponto e finalizadas também por ponto.

Palavras-chave: latex. abntex. editoração de texto.

Abstract

This is the english abstract.

Keywords: latex. abntex. text editoration.

Lista de ilustrações

Lista de tabelas

Lista de códigos

Lista de abreviaturas e siglas

API	Application Program Interface
REST	Representational State Transfer
HTTP	Hypertext Transfer Protocol
HTTPS	Hyper Text Transfer Protocol Secure
ERP	Enterprise Resource Planning

Lista de símbolos

Γ	Letra grega Gama
Λ	Lambda
ζ	Letra grega minúscula zeta
$\in$	Pertence

Sumário

1 Introdução

Em um mundo cada vez mais interconectado, usuários constantemente exigem maior disponibilidade de informações através da Web. Essas informações não devem apenas estarem acessíveis, como tamém é crucial que seja rápido o tempo para acessa-lás. Com a expanção da inernet para dispositivos dos mais diversos tipos, tais como *smartphones*, *tables* e dispositivos de *IoT*, questões como tempo de carregamento e consumo de banda vem se tornando fatores cada mais mais debatidos.

O uso de dispositivos móveis para acessar a internet cresceu 63% somente em 2016. É o que aponta o relatório de Previsão Global de Tráfego de Dados Móveis, elaborada pela Cisco. O tráfego de dados móveis passou de 4.4 exabytes [1] por mês em 2015, para uma média mensal de 7.2 exabytes em 2016. Esse aumento foi resultado de uma adição de cerca 429 milhões de novos dispositivos móveis a rede, sendo *smartphones* os responsáveis pela maior parte deste crescimento (**??**).

Conforme **??**), usuários tendem a dar mais impotancia à velociade em que recebem a informação do que a estética que ela é apresentada. O tempo de carregamento é um fator decisivo para permanência em uma página Web, uma vez que a maioria dos usuários estão dispostos a aguardar de 6 à 10 segundos antes de abandonar qualquer página. Cada segundo de *delay* pode resultar em uma redução de até 7% nas taxas de conversão, o que para um *e-commerce* que fatura mensalmente R$ 100.000,00 por exemplo, pode potencialmente custar R$ 2.5 milhões em vendas não efetuadas em um ano.

Para atender estas novas demandas, nos últimos anos houve uma mudança para um modelo de computação chamado cliente/servidor, que aborda as falhas da computação centralizada. Claramente, o modelo de computação centralizado permanece válido em certos ambientes de negócios, no entanto, apesar de muitos benefícios, a computação centralizada é reconhecida como tendo promovido uma cultura de gerenciamento de informações que não conseguiu atender as necessidades de seus clientes.

Em 2010, ocorreu um grande avanço no número de APIs públicas impulsionado pela transição no modelo de comunicação entre aplicações distribuídas, em que estas passaram a utilizar amplamente o protocolo HTTP e o modelo cliente/servidor para a troca de informações através da Web (**??**). A adoção do REST (REpresentational State Transfer) como o método predominante para construir APIs públicas tem ofuscado qualquer outra tecnologia ou abordagem nos últimos anos. Embora várias alternativas (principalmente SOAP) ainda estejam presentes no mercado, adeptos do modelo SOA para construção de aplicações tomaram uma posição definitiva contra eles e optaram por REST como modelo

[1] Um exabyte é equivalente a um bilhão de gigabytes e mil petabytes.

de comunicação e JSON como seu formato de mensagem (**??**).

Segundo **??**) REST é cada vez mais usado como alternativa ao "já antigo" SOAP em que a principal crítica a este é a burocracia, algo que o REST possui em uma escala muito menor. REST é baseado no *design* do protocolo HTTP, que já possui diversos mecanismos embutidos para representar recursos como código de *status*, representação de tipos de conteúdo, cabeçalhos, etc. O principal nesta arquitetura são as URLs do sistema e os *resources* [2], aproveitando os métodos HTTP para se comunicar.

Entretanto, com o aumento do uso do REST como modelo de comunicação das APIs, algumas limitações foram expostas, prejudicando o desempenho destas APIs em aspectos cruciais. Clientes com rotinas complexas necessitam realizar consultas complexas, buscando objetos aninhados com diversos relacionamentos. Como uma API REST expõe exclusivamente recursos, é necessário executar diversas buscas no servidor para que uma rotina complexa possa ser processada pelo cliente, uma vez que nem todas as informações necessárias estão presentes na resposta de uma única consulta.

Bem como é necessário realizar diversas buscas na API pois uma só consulta pode não contém toda informação necessária, grande parte destas consultas irão retornar dados desnecessárias ao contexto da rotina que executou a busca. Esta prática é conhecida como *over-fetching* e ocorre pois é responsabilidade do servidor da API em montar o conteúdo da resposta, resultando no tráfego de dados desnecessários.

Foram propostas múltiplas soluções para aumentar a eficiência na busca de dados, alguns em relação aos formatos de consulta e resposta das requisições, enquanto outros estão otimizando o número de solicitações na rede. Uma tendência recente envolve um modelo de consultas declarativas de dados, em que as aplicações clientes especificam quais dados precisam, em vez de buscar tudo a partir de um local específico definido por um URL. Estes modelos então otimizaram a comunicação com os servidores para obter os dados de forma eficiente.

Esta é a proposta do GraphQL. Construido pelos desenvolvedores do Facebook para atenter as necessidades internas da rede social em 2012, o GraphQL foi lançado ao público em geral em 2015, e já vem ganhando diversos adeptos. Com a promessa de mitigar alguns problemas crônicos do *design* do REST, como versionamento de APIs, múltiplas viagens de ida e volta e excesso de dados trafegados na rede, a abordagem do Facebook já vem sendo usava por diversas empresas.

REST é de fato, o modelo mais utilizado para comunicação entre cliente e servidor nas aplicações atuais. Este trabalho foca em identificar as diferenças em termos de tempo de carregamento, quantidade de dados trafegados e consumo de recursos entre aplicações REST e o ainda pouco conhecido GraphQL. Ambas as tecnologias são frequentemente

[2] resource é um recurso, entidade

usadas e discutidas como soluções para servir dados entre clientes Web e servidores, com cada vez mais desevolvedores se interessando pela nova abordagem GraphQL

2 Fundamentos

2.1 IPC - Interprocess Communication

Nem sempre um programa sequencial é a melhor solução para um determinado problema. Muitas vezes, as implementações são estruturadas na forma de várias tarefas inter-dependentes que cooperam entre si para atingir os objetivos da aplicação (**??**).

Diversos sistemas operacionais fornecem mecanismos para viabilizar a comunicação e o compartilhamento de dados entre aplicações. Coletivamente, as atividades habilitadas por estes mecanismos são chamadas de interprocess communications (IPC) (**??**).

Os mecanismos que garantem a comunicação entre processos concorrentes e os acessos aos recursos compartilhados são chamados *interprocess communication*. Algumas formas de IPC facilitam a divisão de trabalho entre diversos processos especialistas, enquanto outras facilitam esta divisão entre computadores dentro de uma rede.

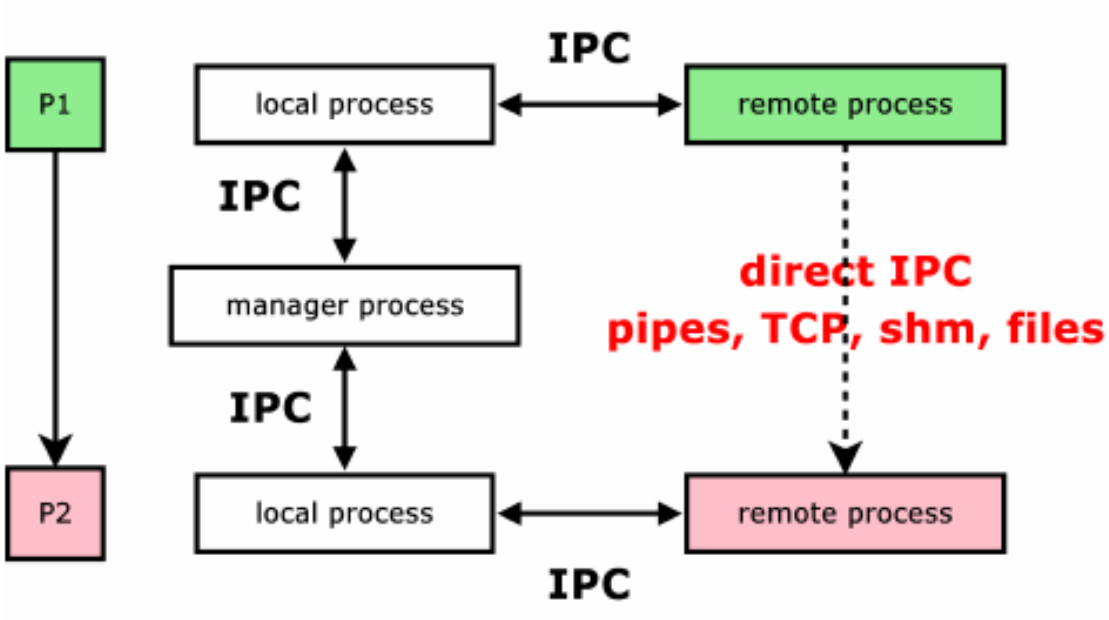
Normalmente, os aplicativos que fazer parte de uma comunicação através de IPC são categorizados como clientes ou servidores. Um cliente é um aplicativo ou um processo que solicita um serviço de alguma outra aplicação ou processo. Por outro lado um servidor é um aplicativo ou um processo que responde a uma solicitação de cliente. Muitas aplicações agem como um cliente e servidor, dependendo da situação (**??**).

A figura **??** mostra como ocorre a comunicação entre processos (P1 e P2). Esta troca de informação pode acontecer de duas maneiras: em duas etapas, ou de forma direta. A comunicação em duas etapas envolve um processo coordenador, que pode ser um interpretador em Python utilizado para dar início ao *workflow* por exemplo. A comunicação direta é ilustrada na figura ligada por linhas pontilhadas e não necessita de intermediação de nenhum processo. Esta maneira de comunicação pode ser executada através de diversas formas, entre elas: Pipes, Shared Memory, Mapped Memory e Arquivos compartilhados.

2.2 Cliente/Servidor

Também conhecido como arquitetura de duas camadas, o modelo cliente/servidor consiste em uma arquitetura em que a camada de apresentação se encontra no cliente e a camada de dados esta armazenada no servidor. Esta separação se opõe ao modelo centralizado amplamente utilizado até seu surgimento.

O processamento dos dados é dividido em duas partes distintas. Uma parte é a requerente de dados (cliente), e a outra parte é a provedora dos dados (servidor). O



Características dos mecanismos de comunicação **??**)

cliente envia durante sua execução uma ou mais solicitações ao servidor para realizar alguma tarefa específica. É dele a responsabilidade tanto de apresentar as informações para o usuário, quando executar as regras de negócio necessárias à aplicação. O servidor é responsável por armazenar os dados e prover um meio para que o cliente os consulte.

Desde a década de 1990, fornecedores de *software* desenvolvem e trazem ao mercado muitas ferramentas para simplificar o desenvolvimento de aplicativos para a arquitetura cliente/servidor de 2 camadas. Algumas das mais conhecidas são: Microsoft Visual Basic, Delphi da Borland e PowerBuilder da Sybase. Estas ferramentas combinadas com milhões de desenvolvedores que sabem usá-las, significam que a abordagem de duas camadas cliente/servidor é uma solução econômica para certas classes de problemas.

Desde então, aplicações *desktop* comunicando-se com o servidor de banco de dados se tornou um caso de uso normal. A maior parte da lógica de negócios foi incorporada dentro da aplicação *desktop*. Portanto, esse estilo de clientes na aquitetura de duas camadas também foi chamado de *fat clients*.

A figura **??** ilustra como a arquitetura em duas camadas funciona, mantendo uma comunicação direta entre cliente e servidor, sem intermediarios entre as duas pontas. Visto que neste modelo as regras de negócio estão presentes na camada de aplicação, ele é frequentemente aplicado em ambientes homogêneos. A camada de banco de dados e a camada de aplicação estão fisicamente próximos, o que oferece um bom desempenho para as aplicações.

Por outro lado, o modelo cliente servidor em duas cadamas possui grandes desafios

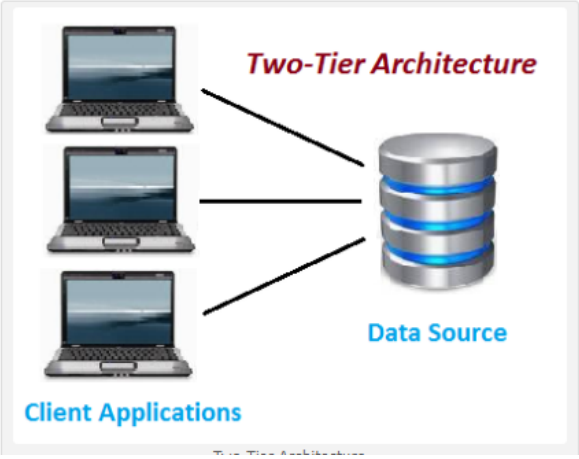


Figura 1 – Comunicação em duas camadas

de escalabilidade. Quando múltiplos usúarios executam requisições simultaneas, a aplicação perde muito desempenho, dado ao fato que cada cliente precisa de conecções separadas e memória de CPU para processar as requisições. Entretando, um dos maiores problemas na arquitetura em duas camadas ocorre quando há mudanças na estrutura do banco de dados. A maioria das aplicações cliente dependem da estrutura do banco de dados impedindo qualquer remodelagem, criando um problema de estruturas legadas, e muitas vezes subutilizadas.

Esta modelo foi substituido pelo modelo de três camadas, muito mais eficiente, e que fornece uma maneira de dividir as funcionalidades envolvidas na manutenção e apresentação de uma aplicação.

2.3 Aplicações Monolíticas

Conforme **??**) explica, uma aplicação monolítica é construída como uma única unidade de *software*. Estas aplicações são construídas em três partes: um banco de dados (que consiste em tabelas geralmente em um sistema de gerenciamento de banco de dados), uma interface de usuário do lado do cliente (consistindo de páginas executando em um navegador ou aplicativos móveis) e um lado do servidor de aplicação. Esta aplicação do lado do servidor tratará as requisições HTTP, executará a regra de negócio, recuperará e atualizará dados do banco de dados e construirá as respostas para serem enviadas às aplicações clientes. Estas camadas representam uma aplicação monolítica - um único executável lógico. Para fazer alterações no sistema, é necessário criar e implantar uma versão atualizada do aplicativo do lado do servidor.

Uma aplicação monolítica é autônoma e independente de outras aplicações. A filosofia do projeto consiste em um aplicativo que não é responsável apenas por uma determinada tarefa, mas que também pode executar todos os passos necessários para

completar uma determinada função, como é mostrado na figura **??**.

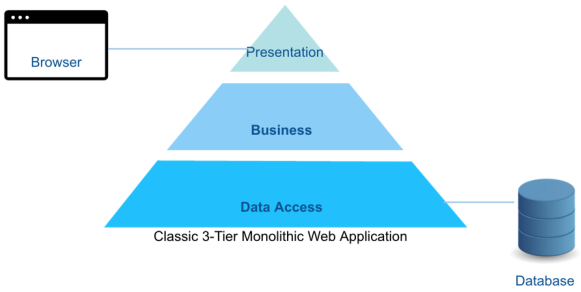


Figura 2 – Aplicação monolítica clássica em 3 camadas

??) afirma que as aplicações monolíticas são uma maneira natural para a evolução de uma aplicação. A maioria das aplicações começam com um único objetivo, ou um pequeno número de objetivos relacionados. Ao longo do tempo, novas funcionalidades são adicionados ao aplicativo para suportar as necessidades do negócio. Entretando, as aplicaçõies monolíticas apresentam algumas desvantagens, sendo alguns deles:

- Um único ponto de falha pode comprometer o funcionamento correto de todos os módulos do sistema.
- Baixa estalabilidade dado a necessidade de copiar toda a *stack* para escalar horizontalmente.
- Base de código **gigante**, uma vez que toda a regra de negócio se encontra em uma só base.
- É necessário muita comunicação para que várias equipes de desenvolvedores trabalhem em paralelo. Esta sobrecarga diminui o ritmo de desenvolvimento.

Contudo, com a popularização dos *frameworks* Javascript como Angular e Backbone, houve um movimento de desacoplamento da camada de visualização nas aplicações monolíticas. Foram criadas então as chamadas aplicações *frontend*, responsáveis pela camada de apresentação, e sendo executadas em dispositivos móveis como *smartphones* e *tables* ou nos *browsers desktop*.

Esse movimento criou um modelo monolítico de duas camadas, ilustrado na figura **??**. A remoção da interdependência entre as camadas de interface e servidor de aplicação facilitou o escalonamento e a escalabilidade de cada uma das partes. Esta separação de conceitos foi o primeiro passo em direção a uma arquitetura orientada a serviços.

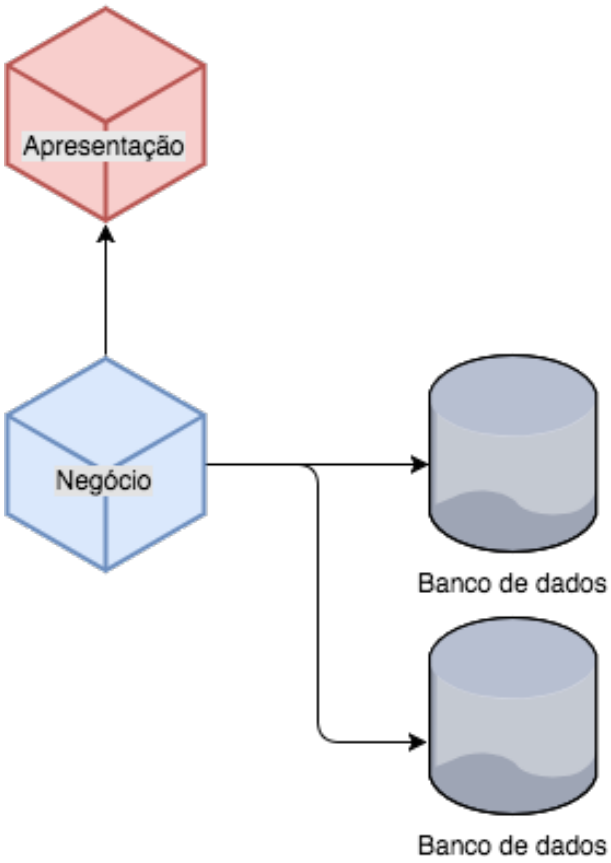


Figura 3 – Modelo monolítico de duas camadas

2.4 SOA - Service Oriented Architecture

??) define SOA como políticas, práticas e frameworks que permitem que rotinas de aplicações sejam fornecidas e consumidas como conjuntos de serviços publicados em uma granularidade relevante para o consumidor do serviço. Os serviços podem ser invocados, publicados e descobertos e são abstraídos da implementação usando uma única forma de interface baseada em padrões.

Além de criar e expor serviços, **??**) assinala que o SOA tem a capacidade de aproveitar esses serviços de forma recursiva em aplicações (conhecidos como aplicativos compostos). O SOA vincula esses serviços à orquestração ou alavanca individualmente esses serviços. Desta forma, o SOA é visto como uma evolução das arquiteturas existentes, abordando a maioria dos principais sistemas como serviços e resgatando esses serviços em um único domínio onde eles formam soluções.

O SOA permite a reutilização de ativos existentes, onde novos serviços podem ser criados a partir de uma infra-estrutura de sistemas de TI existente. Em outras palavras, ele permite que as empresas reutilizem aplicativos existentes e possibilita a interoperabilidade entre aplicações e tecnologias heterogêneas. Ele fornece um nível de flexibilidade que não era possível antes, no sentido de que:

- Os serviços são componentes de software com interfaces bem definidas que são independentes da implementação. Um aspecto importante do SOA é a separação da interface de serviço (o que) da sua implementação (como). Esses serviços são consumidos por clientes que não estão preocupados com a forma como esses serviços irão executar suas solicitações;
- Os serviços são autônomos (executar tarefas predeterminadas) e vagamente acoplados (para independência);
- Serviços compostos podem ser construídos a partir de agregados de outros serviços;

Isso significa que o SOA é uma abordagem alinhada com o negócio, em que os aplicativos dependem dos serviços disponíveis para facilitar os processos de negócios. Um serviço é um componente de *software* reutilizável e autônomo, fornecido por um provedor de serviços e consumido pelos solicitantes. O SOA cria uma visão de flexibilidade de TI que facilita a agilidade do negócio. Sua implementação envolve principalmente componentes de aplicativos corporativos e/ou em desenvolvimento que usam serviços, disponibilizando aplicativos como serviços para outras aplicações (**??**).

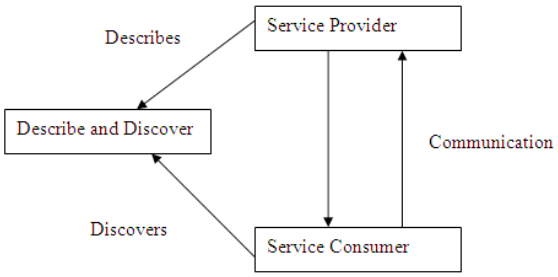


Figura 4 – Service Oriented Architecture

??) acrescenta que em um ambiente SOA típico, existe um provedor de serviços e um consumidor de serviços. Para que isso funcione, também é necessário um mecanismo para que eles possam se comunicar uns com os outros, como é ilustrado na figura **??**. A W3C [1] definiu um padrão aberto para que *web services* possam implementar o SOA e habilitar a comunicação entre o provedor e o consumidor através de um protocolo baseado em XML: o *Simple Object Access Protocol* (SOAP). Outros padrões também são usandos ou foram criados para realizar a comunicações entre os serviçoes SOA. Alguns basearam-se no *design* do *REpresentation State Transfer* (REST), utilizando tanto XML quando JSON para transportar os dados entre os serviços.

Em geral, há uma confusão entre o relacionamento entre SOA e *web services*. **??**) faz uma distinção entre o SOA e web services do seguinte modo: Ẅeb services tratam-se

[1] World Wide Web Consortium

de especificações de tecnologias, enquanto o SOA é um princípio de *design* de *software*. Notavelmente, Web services são um padrão de definição de interface adequado ao SOA e é neste ponto onde eles se conectam ao SOA". Portanto, o SOA é um padrão de arquitetura, enquanto os *Web services* são serviços implementados usando um conjunto de padrões. Os *Web services* são uma das maneiras em que é possível implementar o SOA. cujo os benefícios são que é possível alcançar uma abordagem neutra em relação a plataformas para acessar serviços e uma melhor interoperabilidade à medida que mais fornecedores suportam cada vez mais especificações.

3 Protocolos de Comunicação

Neste capítulo será apresentado alguns dos principais protocolos para a comunicação entre aplicações. Serão conduzidas breves abordagens dos protocolos RPC e SOAP, que hoje vem perdendo espaço no mercado, mas tiveram um papel muito importante para a evolução dos sistemas SOA. Em seguida, será mostrada uma descrição mais detalhada dos modelos de comunicação REST e GraphQL, uma vez que estes são os dois modelos a serem seguidos no estudo de caso.

Será apresentado as principais características desses protocolos, como também suas limitações. Nenhum dos protocolos podem ser descardado na hora de tomada de decisão sobre qual modelo utilizar para a comunicação entre clientes e serviços remotos ou APIs. Todos eles possuem limitações, porém cumprem bem o papel para que foram desenhados e é possível notar como os mesmos evoluiram a medida que novas demandas se tornarão necessária no desenvolvimento de software ao decorrer do tempo.

3.1 Remote Procedure Call

Remote Procedure Call (RPC), ou Chamada de Procedimento Remoto, é um mecanismo em que uma aplicação solicita o serviço de uma outra aplicação que se encontra em um outro computador, geralmente conectados por uma rede. Um RPC exige que uma aplicação X envie uma ou mais mensagens para outra aplicação Y a fim de invocar um procedimento da aplicação Y. A aplicação Y responde enviando uma ou mais mensagens de volta para a aplicação X. O termo RPC identifica todo o processo de comunicação entre as aplicações (**??**).

O RPC é similar ao modelo de chamadas de procedimentos locais (IPC). Nas chamadas locais, a aplicação remetente envia argumentos para um procedimento localização no mesmo computador e transfere o controle da CPU para o procedimento. Logo após o termido da execução, os resultados do procedimento são extraídos de uma localização compartilhada, e a aplicação remetente continua sua execução (**??**). As chamadas remotas por outro lado, aguardam a os resultados do procedimento diretamente da aplicação requisitada, como é ilustrado na figura **??**

Ainda, segundo a especificação do RPC (**??**), há alguns aspectos importantes em que as chamadas remotas se diferem das chamadas locais:

a **Tratamento de erros**: Falhas no servidor remoto ou da rede devem ser tratadas quando usando uma RPC.

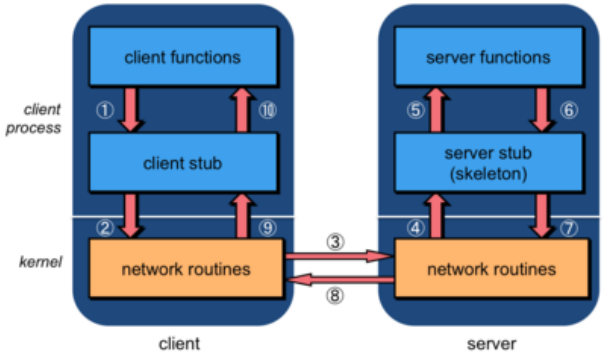


Figura 5 – RPC fluxo de dados.(refazer)

b **Variáveis globais e efeitos colaterais**: Uma vez que o servidor não tem acesso ao espaço de endereço do cliente, argumentos ocultos não podem ser passados como variáveis globais ou retornados como efeitos colaterais.

c **Desempenho**: Chamadas remotas geralmente operam uma ou mais ordens mais lentos do que chamadas de procedimentos locais

d **Autenticação**: Uma vez que as chamadas remotas pode ser transportadas através de redes insegurar, autenticação pode ser necessário. Autenticação previne uma entidade de se passar por outra e executar chamadas indevidas.

3.1.1 Limitações

Diferentes implementações de RPC são, geralmente incompatíveis. Assim sendo, o uso de uma implementação específica provavelmente resultará na dependência da implementação do fornecedor. Esse tipo de incompatibilidade entre diferentes implementações implica em um diverso número de funcionalidades, suportando inúmeros protolos de rede e diferentes sistemas de arquivos.

??) aponta diversas limitações técnicas e de desempenho encontradas em implementações de RPC's. Dentre estas limitações, se destacam:

- **Organização dos parâmetros**: para organizar os parâmetros, o cliente precisa conhecer a quantidade e a tipagem exata dos parâmetros. Como não existe um padrão bem definido, a estruturação dos parâmetros pod variar entre diferentes implementações.
- **Falta de paralelismo**: com RPC, em um determinado momento, ou o servidor ou o cliente está ativo. Como cliente e servidor são co-rotinas, a prática de paralelismo presente em outros modelos de comunicação, não é possível de ser implementada.
- **Ferramentas de padronizaçào**: as ferramentas padronizadas tornam-se muito mais difíceis de construir. Construir estas ferramentas geralmente requer o redesenho

de convenções de tipo HTTP baseado sistema RPC escolhido, por consequencia da especialidade de cada sistema RPC.

3.2 Simple Object Access Protocol

Simple Object Access Protocol (SOAP) é um protocolo para troca de mensagens em ambientes descentralizados e distribuídos. **??**) definem o SOAP como um protocolo baseado em XML que consiste em três partes: um envelope que defina um *framework* para descrever o conteúdo das mensagens trafegadas e como processar este conteúdo, um conjunto de regras de codificação para expressar instâncias de tipos de dados definidos por uma aplicação, e uma convenção para representar chamadas remotas e suas respostas.

??) argumenta que o SOAP pode ser considerado parecido com as Chamadas de Procedimento Remoto (RPC), porém eliminando algumas das complexidades frequentemente encontradas nas implementações de RPCs. Utilizando o SOAP, é possível chamar serviços de outras aplicações que estão sendo executadas em qualquer *hardware*, independente do sistemas operacional ou da linguagem de programação.

Embora a especificação do SOAP tenha evoluído para longe da necessidade de acessar objetos, como ocorre com as Chamadas de Procedimentos Remoto, existem ainda convenções para o encapsulamento e envio de chamadas RPC utilizando uma representação uniforme de chamadas e repostas. Definindo um padrão para o mapeamento das chamadas RPC para SOAP, torna-se possível para infraestrutura traduzir as invocações de serviços em mensagens com o protocolo SOAP em tempo de execução, sem a necessidade de redesenhar todo o serviço Web presente na plataforma (**??**).

A figura **??** ilustra como é realizada a troca de mensagens entre uma aplicação cliente e o servidor utilizando o SOAP. Como é possível observar, a aplicação cliente primeiramente serializa a mensagem e envia. Ao receber a requisição, o servidor precisa decerializar e mensagem, executa a consulta e serializa a mensagem de resposta. A aplicação cliente então, deserializar a resposta para poder manipulá-la.

??) assinala que SOAP é um protocolo que define uma gramática XML especializada, que padroniza o formato das estruturas das mensagens. As mensagens são, por outro lado, o método fundamental de troca de informações entre os serviços Web e os seus consumidores. Ao utilizar XML para codificar mensagens,o SOAP apresenta alguns benefícios:

a Permite a comunicação entre sistemas protegidos por *firewalls*, sem precisar abrir portas adicionais e possivelmente não seguras;

b É mais fácil de entender e eliminar erros pois o formato XML pode ser facilmente lido por humanos;

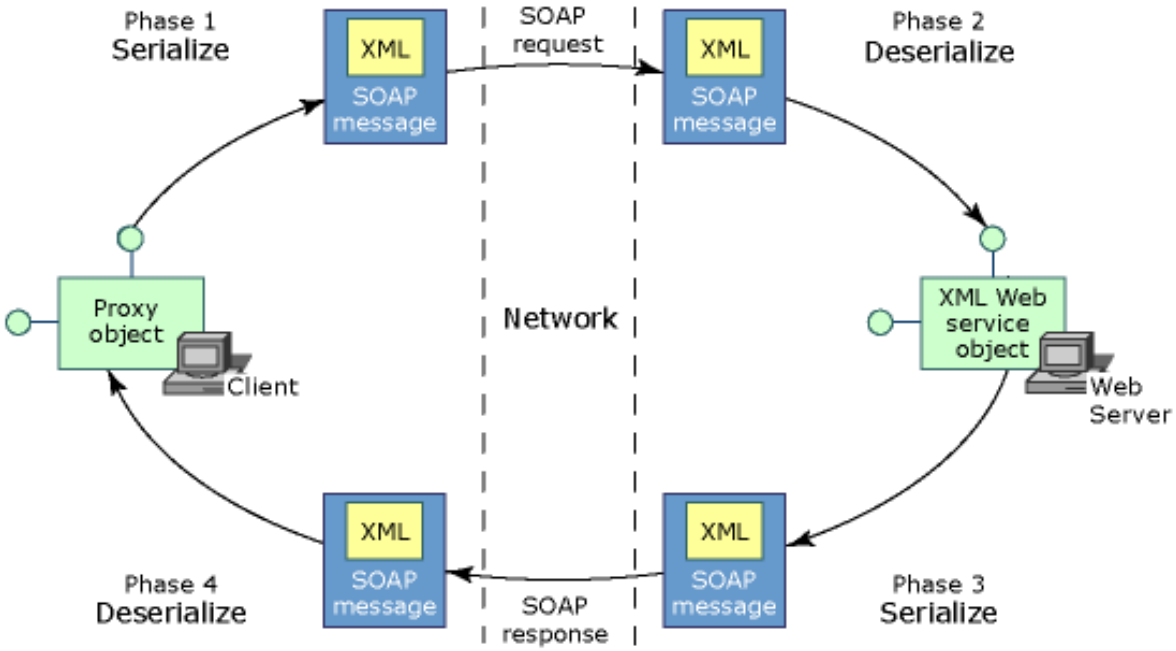


Figura 6 – SOAP fluxo de dados. (refazer)

c As mensagens podem ser compreendidas por quase todas as plataformas de hardware, sistemas operacionais e linguagens de programação uma vez que os dados do SOAP são estruturados usando XML.

d Pode ser usado, potencialmente, em combinação com vários protocolos de transporte de dados, como HTTP, SMTP e FTP.

e O SOAP mapeia satisfatoriamente para o padrão de solicitação / resposta do HTTP.

3.2.1 Limitações

??) explica que tecnologias de metadados abertas, como o XML, podem fornecer um grande ganho de usabilidade, mas o sucesso dessas tecnologias exige que seu uso não degrada o desempenho de forma irracional.O XML é extremamente robusto, no entanto, o seu uso pode afetar negativamente o desempenho do SOAP nas seguintes áreas:

- velocidade de codificação e decodificação. A conversão de dados de binário para ASCII e vice-versa é o principal custo de desempenho do XML. O uso de um protocolo baseado em texto também impede a aplicação de otimizações disponíveis para protocolos binários quando a comunicação ocorre entre sistemas homogêneos.
- tamanho da mensagem. Para XML, um fator de expansão de 6-8 vezes em relação aos dados binários originais não é incomum e que o tamanho da representação de dados do SOAP é tipicamente cerca de 10 vezes o tamanho da representação binária

equivalente. Este tamanho substancialmente maior pode resultar em maiores custos de transmissão de rede e latência aumentada.

Como é possível notar, as maiores limitações encontradas na utilização do SOAP como modelo de comunicação estão relacionadas a estruturação usando XML. Embora esse formato padrão de mensagens tenha trazido vantagens em relação às tecnologias anteriores, ele também trouxe uma série de limitações conforme sua popularidade foi crescendo. A necessidade buscar alternativas para superar esses problemas foi crescente, e novos formatos foram aparecendo, tendo como destaque o JSON.

3.3 REpresentational State Transfer

REpresentational State Transfer (REST) é um *design* de arquitetura baseado em um conjunto de princípios que descrevem como os recursos em rede são definidos e abordados. Estes princípios foram descritos pela primeira vez em 2000 por Roy Fielding, como parte de sua dissertação de doutorado (**??**).

??) assinala que a adoção do REST como o método predominante para construir APIs públicas tem ofuscado qualquer outra tecnologia ou abordagem nos últimos anos. Embora várias alternativas (principalmente SOAP) ainda estejam presentes no mercado, adeptos do modelo SOA, descrito da seção 2.4, para construção de aplicações tomaram uma posição definitiva contra eles e optaram por REST como sua abordagem e JSON como seu formato de mensagem .

Segundo **??**), REST é um padrão de operações de recursos que emergiu como a principal alternativa ao SOAP para o *design* de serviços em aplicativos Web 2.0. Considerando que a abordagem tradicional baseada em SOAP para Web Services usa objetos remotos completos com invocação de método remoto e funcionalidade encapsulada, o REST trata apenas de estruturas de dados e da transferência de seu estado. A simplicidade do REST, juntamente com seu ajuste natural sobre o HTTP, contribuiu para o seu status como um método de escolha para aplicativos da Web 2.0 expondo seus dados.

3.3.1 Restrições

As restrições do REST são regras desenhadas para estabelecer as características distintas da arquitetura REST. Cada restrição é uma decisão de projeto pré-determinada que pode ter impactos positivos e negativos. A intenção é que os aspectos positivos de cada restrição equilibrem os negativos para produzir uma arquitetura geral que se assemelhe à Web.

Uma arquitetura que elimina uma das restrições de **cliente/servidor**, **stateless**, **cache**, **interface uniforme**, **sistemas em camadas** ou **código por demanda**, geral-

mente é considerada como não mais conforme ao REST. Isso exige que as decisões sejam tomadas para entender os potenciais *trade-offs* quando se desviam deliberadamente da aplicação de restrições REST.

A primeira restrição estabelecida é a da arquitetura **cliente-servidor**, descrita na Seção 2.2. A separação de responsabilidades é o princípio por trás das restrições cliente-servidor. Ao separar as responsabilidades da interface do usuário com as responsabilidades de armazenamento de dados, é melhorada a portabilidade da interface do usuário em várias plataformas e a escalabilidade, simplificando os componentes do servidor. Talvez o mais importante para a Web, no entanto, é que a separação permite que os componentes evoluam de forma independente, apoiando assim o requisito de escalabilidade da Internet de múltiplos domínios organizacionais (**??**)

Em seguida, foi adicionado ao REST uma restrição à interação cliente-servidor: a comunicação deve ser totalmente **stateless**, ou seja, independente de estado, de modo que cada solicitação do cliente para o servidor deve conter todas as informações necessárias para entender o pedido e não pode tirar proveito de nenhum contexto armazenado no servidor. O estado da sessão é, portanto, mantido inteiramente no cliente (**??**).

Essa restrição induz as propriedades de visibilidade, confiabilidade e escalabilidade. A visibilidade é melhorada porque um sistema de monitoramento não precisa olhar além de um único dado de solicitação para determinar a natureza completa da solicitação. A confiabilidade é melhorada porque facilita a tarefa de recuperação de falhas parciais. A escalabilidade é melhorada porque não ter que armazenar o estado entre solicitações permite que o componente do servidor rapidamente libere recursos e simplifica ainda mais a implementação porque o servidor não precisa gerenciar o uso de recursos em todos os pedidos.

A restrição de **cache** foi criada para melhorar a eficiência da rede. Ela exige que os dados dentro de uma resposta a uma solicitação sejam rotulados de forma implícita ou explícita como cacheáveis ou não armazenáveis em cache. Se uma resposta for armazenada em cache, um cache do cliente terá o direito de reutilizar esses dados de resposta para solicitações equivalentes posteriores.

A vantagem de adicionar restrições de cache é que eles têm o potencial de eliminar parcial ou totalmente algumas interações, melhorar a eficiência, escalabilidade e desempenho percebido pelo usuário, reduzindo a latência média de uma série de interações. O *trade-off*, no entanto, é que um cache pode diminuir a confiabilidade se os dados obsoletos dentro do cache diferirem significativamente dos dados que teriam sido obtidos se a solicitação fosse enviada diretamente para o servidor.

A característica central que distingue a arquitetura REST de outros estilos baseados em rede é a ênfase em uma **interface uniforme** entre os componentes. Ao aplicar o

princípio de engenharia de software de especialização/generalização para a interface do componente, a arquitetura geral do sistema é simplificada e a visibilidade das interações é melhorada. As implementações são dissociadas dos serviços que eles fornecem, o que incentiva a evolução independente. O trade-off, no entanto, é que uma interface uniforme degrada a eficiência, uma vez que a informação é transferida em uma forma padronizada e não específica para as necessidades de uma aplicação.

Para obter esta interface uniforme, são necessárias várias restrições arquitetônicas para orientar o comportamento dos componentes. REST é definido por quatro restrições de interface: identificação de recursos; Manipulação de recursos através de representações; Mensagens auto-descritivas; E, hipermídia como motor do estado da aplicação.

A fim de melhorar o comportamento dos requisitos de escalonamento, o REST também implementa restrições de **sistema em camadas**. O sistema em camadas permite que uma arquitetura seja composta de camadas hierárquicas ao restringir o comportamento dos componentes, de modo que cada componente não possa acessar além da camada imediata com a qual eles estão interagindo. Ao restringir o conhecimento do sistema a uma única camada, colocamos um limite na complexidade geral do sistema e promovemos a independência do componentes. As camadas podem ser usadas para encapsular serviços legados e para proteger novos serviços de clientes legados, simplificando componentes e movendo funcionalidades raramente usadas para um intermediário compartilhado. Os intermediários também podem ser usados para melhorar a escalabilidade do sistema ao permitir o balanceamento de carga de serviços em várias redes e processadores.

A principal desvantagem dos sistemas em camadas é que eles adicionam sobrecarga e latência ao processamento de dados, reduzindo o desempenho perceptível pelo usuário. Para um sistema baseado em rede que suporta restrições de cache, isso pode ser compensado pelos benefícios do armazenamento em cache compartilhado em intermediários.

A ultima restrição que o REST implementa é a restrição de **código por demanda**. O REST permite que a funcionalidade do cliente seja estendida baixando e executando o código na forma de *applets* ou *scripts*. Isso simplifica os clientes, reduzindo o número de recursos necessários para serem pré-implementados. Permitir que os recursos sejam baixados após a implantação, melhora a extensibilidade do sistema, no entanto, também reduz a visibilidade e, portanto, é apenas uma restrição opcional dentro do REST.

3.3.2 Modelo de Maturidade de Richardson

O Modelo de Maturidade de Richardson é um modelo criado por Leonard Richardson que descreve os requisitos necessários para o desenvolvimento de uma API REST bem estruturada e compatível com as restrições definidas pela arquitetura REST. Quanto melhor a API adere às restrições citadas na sessão anterior, melhor será pontuada. O modelo de

Richardson, ilustrado na figura **??**, descreve 4 níveis (0-3), onde o nível 3 designa uma API verdadeiramente *RESTful*. Richardson usou três fatores para decidir a maturidade de uma API: URI (*Uniform Resource Identifiers*), métodos HTTP e HATEOAS (Hypermedia). Quanto mais uma API emprega essas tecnologias, mais madura ela é categorizada.

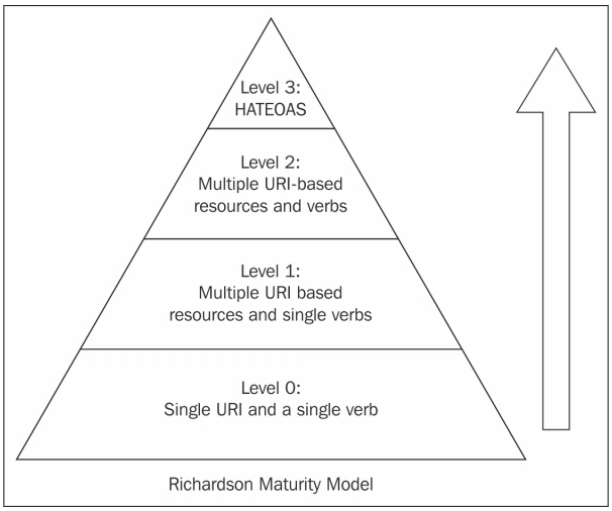


Figura 7 – Modelo de maturidade de Richardson. (refazer)

- Nível 0: O ponto de partida para o modelo é usar HTTP como um sistema de transporte para iterações remotas, mas sem usar nenhum dos mecanismos da web. Essencialmente, o que está sendo feito é utilizar o HTTP como um mecanismo de tunelamento para seu próprio mecanismo de interação remota, geralmente com base em Chamadas de Procedimento Remotas.

  É como se estivesse sendo chamado funções, porém através do uso do HTTP. Todos os serviços são centralizados em um único *endpoint*, ou seja, todas as solicitações são feitas em uma única URI.

- Nível 1: Quando uma API faz a distinção entre recursos diferentes, pode se considerar que atingiu o nível 1. Esse nível usa vários URIs, em que cada URI é o ponto de entrada para um recurso específico. Ainda assim, esse nível usa apenas um único método, como o POST.

- Nível 2: Esse nível indica que a API deve usar as propriedades do protocolo para lidar com escalabilidade e falhas. Não é recomendado que se use um único método POST para todas as chamadas, mas faça uso do GET quando estiver solicitando recursos e use o método DELETE quando desejar excluir recursos. Além disso, o uso dos códigos de resposta do protocolo de aplicação também é recomendado. Não deve ser usando, por exemplo, o código 200 (OK) quando algo der errado.

- Nível 3: O nível três de maturidade faz uso de todos os três fatores, isto é, URIs, métodos HTTP e HATEOAS. Esse é o nível em que a maioria das APIs menos implementam e pois não seguem o princípio de HATEOAS.

  HATEOAS (Hypermedia as the Engine of Application State) é uma abordagem para a construção de serviços REST, em que o cliente pode descobrir dinamicamente as ações disponíveis em tempo de execução. Todo cliente deve exigir uma URI inicial e um conjunto de tipos de mídia padronizados para começar a troca de mensagens. Uma vez que carregou o URI inicial, todas as futuras transições de estado da aplicação serão conduzidas pelo cliente selecionando as escolhas fornecidas pelo servidor.

  Não seguir essa abordagem não é necessariamente ruim. Há alguns bons argumentos a serem feitos a favor e contra a utilização de HATEOAS. Enquanto, por um lado, torna as APIs fáceis de descobrir e usar, que geralmente vem ao custo de mais tempo e esforço de desenvolvimento.

3.3.3 Limitações

Contudo, com a evolução e o aumento da complexidade das APIs, a comunicação via o protocolo REST vem se mostrando muitas vezes inviável pois sua implementação trás como consequências alguns problemas estruturais como:

- A necessidade de executar múltiplas requisições entre o cliente e o servidor a fim de obter objetos complexos e com atributos aninhados Para aplicações móveis operando em condições variáveis de rede, essas múltiplas viagens de ida e volta são indesejáveis, pois geram atrasos e maior tráfego na rede (**??**).

- A prática de *over-fetching*, ou seja, quando o cliente busca alguma informação do servidor e a resposta contém mais informação que o cliente precisa. Esse tipo de problema acarreta no uso desnecessário de recursos de comunicação (**??**).

- O versionamento da API, que ocorre quando há alterações significativas na API, sujeitas a quebra de código nos clientes consumidores. Essa prática pode ser extremamente penosa se a API é usada por uma grande massa de clientes que não são facilmente atualizados (**??**).

3.4 Arquiteturas baseadas em JSON/Graphs

Recentemente, um novo design de arquitetura vem ganhando espaço, preenchendo algumas lacunas que arquiteturas anteriores deixaram. Lideradas pelo Facebook com seu GraphQL e Netflix com o Falcor, esta nova arquitetura da um passo para trás comparando-se ao REST, atingindo apenas o nivel 0 no Modelo de Maturidade de Richardson.

Os principais problemas que essa arquitetura ajuda a resolver são: A dependência das aplicações cliente nos servidores, eliminando a necessidade do servidor de ter que manipular as informações ou o tamanho da resposta; A necessidade de executar diversas requisições para acessar os dados exigidos por uma *view*; Melhorar a experiência de desenvolvimento *frontend*, pois existe uma relação próxima entre os dados necessários à interface da aplicação e a forma como um desenvolvedor pode expressar uma descrição desses dados para a API. O presente trabalho irá apenas focar no GraphQL como representante das arquiteturas baseadas em JSON/Graph.

GraphQL é uma linguagem de consulta, criada pelo Facebook em 2012, que fornece uma interface comum entre o cliente e o servidor para manipulação e busca de dados. GraphQL utiliza de um sistemas chamado *cliente-specified queries*, onde o o formato de resposta de uma requisição é definida pelo cliente. **??**) afirma que uma vez que a estrutura de dados não é codificada, como nas APIs tradicionais, a consulta de dados do servidor se torna mais eficiente para o cliente.

Consultas utilizando GraphQL sempre retornam apenas o que foi pré definido pela requisição, fazendo suas respostas sempre serem previsíveis. As consultas com GraphQL acessam não apenas as propriedades de um único recurso, mas também seguem as referências entre eles. Enquanto as APIs REST típicas exigem o carregamento de múltiploas URLs, as APIs GraphQL obtêm todos os dados que precisam em uma única requisição.

Adicionar novos campos ou tipos à uma API GraphQL não afeta nenhuma consulta ou funcionalidade já existente. Campos não mais utilizados podem ser obsoletos e ocultos de ferramentas de mapeamento. Ao usar uma única versão em evolução, as APIs GraphQL dão às aplicações acesso contínuo a novos recursos e encorajam um código de servidor mais limpo e mais sustentável.

3.4.1 Operações

Existem três tipos de operações modeláveis com GraphQL:

- Query: Uma consulta de somente leitura;
- Mutation: Uma escrita seguida por uma consulta;
- Subscription – Uma requisição de longa duração que obtém dados em resposta a eventos disparados pelo servidor;

Uma Query em GraphQL é uma maneira de obter dados de uma maneira somente de leitura em uma API GraphQL. De uma maneira geral, GraphQL se baseia em consultar campos específicos em objetos. Isso significa que a consulta tem exatamente o mesmo

formato que a resposta. Isso é essencial para GraphQL, porque a resposta é sempre previsível, e o servidor sabe exatamente quais os campos que o cliente está pedindo.

Como GraphQL não se limita apenas em consultas de dados, as APIs também podem implementar operações para criar, atualizar e destruir dados. Para esses tipos de operações, o GraphQL usa o termo Mutations. As Mutations são uma maneira de alterar dados em seu servidor. É importante notar que as *mutations* consistem em uma alteração seguida de uma busca do dado que acabou de ser alterado, tudo em uma única operação.

Os aplicativos em tempo real precisam de uma maneira de enviar dados do servidor. As *subscriptions* permitem que aplicações publiquem eventos em tempo real através de um servidor de assinaturas GraphQL. Com o modelo de subscriptions baseado em eventos no GraphQL - muito parecido com as *queries* e *mutations* - um cliente pode dizer ao servidor exatamente quais dados devem ser enviados e como esses dados devem ser encontrados. Isso leva a menos eventos rastreados no servidor e notificados para o cliente.

3.4.2 Schemas e Tipos

O sistema de tipos do GraphQL descreve as capacidades de um servidor GraphQL e é usado para determinar se uma consulta é válida. O sistema de tipos também descreve os formatos de entrada de variáveis de consulta para determinar se os valores fornecidos em tempo de execução são válidos. As capacidades do servidor GraphQL são referidas como *schema* do servidor. Um *schema* é definido baseado nos tipos que ele suporta.

Cada servidor GraphQL define um conjunto de tipos que descrevem completamente o conjunto de dados possíveis que você pode consultar nesse servidor. Então, quando as consultas chegam, elas são validadas e executadas contra os *schemas*. O *schema* descreverá quais campos o servidor pode responder e quais tipos de objetos estarão contidos nas respostas. A informação de tipo é muito importante para o GraphQL e o cliente pode assumir de forma segura que o servidor retornará tipos consistentes de objetos para o mesmo campo.

A unidade fundamental de qualquer *schema* GraphQL é o tipo. Existem oito formatos de tipos suportados no GraphQL. O tipo mais básico é o ***Scalar***. Um campo do tipo *Scalar* representa um valor primitivo, como uma *string* ou um número inteiro. Muitas vezes, as respostas possíveis para um campo do tipo *Scalar* são enumeráveis. O GraphQL oferece um tipo ***Enum*** nesses casos, onde o tipo especifica o espaço de respostas válidas. Campos do tipo ***Object*** definem um conjunto de campos, onde cada campo é outro formato, permitindo a definição de hierarquias de tipos arbitrários. O GraphQL suporta dois tipos abstratos: ***Interfaces*** e ***Unions***.

Todos os tipos até agora são considerados nulos e singulares, por exemplo, uma *string* retorna um valor nulo ou singular. O sistema de tipo pode querer definir que ele

retorna uma lista de outros tipos. O tipo ***List*** é fornecido por esse motivo e envolve outro tipo. Da mesma forma, o tipo ***Non-Null*** envolve outro tipo e indica que o resultado nunca será nulo.

Finalmente, muitas vezes é útil fornecer estruturas complexas como entradas para consultas GraphQL; o tipo ***Input Object*** permite que o *schema* defina exatamente quais dados são esperados do cliente nessas consultas.

3.4.3 Autorização

Segundo **??**), Autorização é o ato de determinar se um direito particular, como acesso a algum recurso, pode ser concedido a uma determinada credencial. Nos sistemas de informação multiusuários, o administrador do sistema define quais usuários podem ter acesso ao sistema e quais são os privilégios de uso.

Normalmente, o GraphQL não controla diretamente o controle de acesso, em vez disso, delega essa responsabilidade para a camada de lógica de negocio da aplicação.

3.4.4 Caching

De acordo com **??**), consultas de dados através da internet podem ser lentas e custosas, e por esse motivo, a capacidade de armazenar em cache e reutilizar os recursos obtidos anteriormente é um aspecto crítico da otimização para o desempenho.

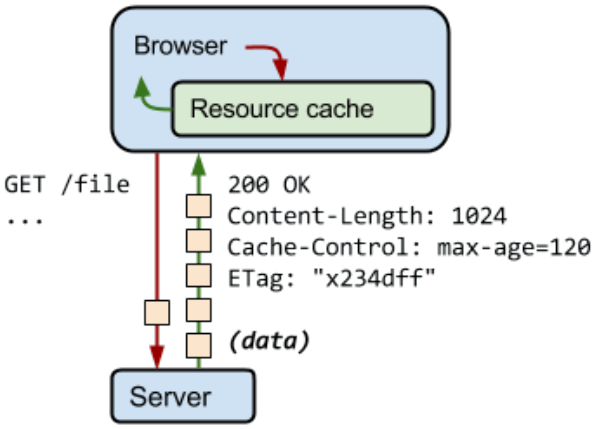


Figura 8 – HTTP Caching. (refazer)

Em uma API baseada em *end-points*, os clientes podem usar o armazenamento em cache do protocolo HTTP, ilustrado na figura **??** para evitar buscas desnecessárias identificando quando dois recursos são iguais. A URL nessas APIs é o identificador globalmente exclusivo que o cliente pode aproveitar para criar um cache. No GraphQL, porém, não existe uma primitiva semelhante a uma URL que forneça esse identificador

globalmente exclusivo para um determinado objeto. Entretanto, existem várias práticas que podem ser aplicadas a um servidor GraphQL para atingir o mesmo objetivo.

Um padrão possível para isso é reservar um campo, como id, para ser o identificador globalmente exclusivo. Da mesma forma que os URLs de uma API baseada em recursos fornecem uma chave globalmente exclusiva, o campo id neste sistema fornece uma chave globalmente única. Outra abordagem possível funcionaria de forma semelhante ao padrão usada nas APIs baseadas em *end-points*. O texto da consulta em si pode ser usado como identificador globalmente exclusivo.

3.4.5 Limitações

Segundo aponta **??**), uma ameaça importante que o GraphQL *facilita* são os ataques de negação de serviço. Um servidor GraphQL pode ser atacado com consultas excessivamente complexas que irão consumir todos os recursos do servidor. Esse tipo de ataque não é específico do GraphQL, mas é preciso um cuidado redobrado para evitar-los.

Há entretanto, alguns procedimento que se implementados, podem mitigar a ameaça de negação de serviço. É possível fazer uma análise de custos na consulta com antecedência e impor *um certo tipo* de limites na quantidade de dados que uma requisição pode consumir. Também é possível implementar um tempo limite para que requisições que levam muito tempo para serem resolvidas, sejam excluídas da fila de execução .

4 Estudo de Caso

Para obter um exemplo do mundo real, um estudo de caso foi projetado com base em uma aplicação de WMS (*Warehouse Management System*). **??**) explica que um sistema de WMS é um software que auxilia as operações do dia-a-dia em um armazém. Os sistemas WMS permitem o gerenciamento centralizado de tarefas, como o rastreamento de níveis de inventário e locais de estoque. Tais sistemas WMS podem ser um aplicativo independente ou parte de um sistema de ERP.

Baseado em um sistema de WMS, dois protótipos de APIs serão implementados para uma analise de desempenho comparando-as. A primeira API será implementada seguindo as melhores práticas do *design* do REST, descritos na sessão X. A segunda será desenvolvida utilizando o GraphQL como motor de resposta, e seu desempenho será comparado com a API REST.

Com o intuito de realizar consultas relevantes, foi modelado um esquema contendo seis entidades capazes de representar consultas reais, e ainda produzirem informações pertinentes para a comparar o desempenho dos protótipos. A construção das APIs seguiram os princípios do ciclo tradicional do desenvolvimento de software, portanto a modelagem foi elaborada antes do início da implementação das APIs. Isso assegura que nenhuma decisão de tecnologia tenha afetado a modelagem das entidades.

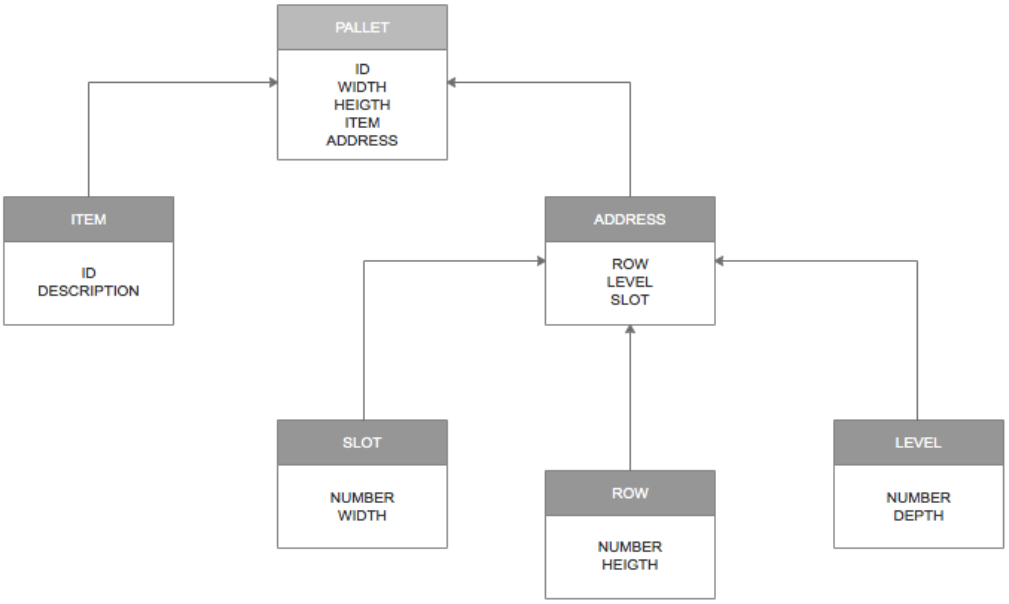


Figura 9 – Modelagem WMS

A modelagem da figura **??**, representa o gerenciamento de um armazém, com

capacidade de armazenar diversos itens. Os itens são dispostos em *pallets*, e alocados em endereços dentro do armazém. Como pode ser observado na figura **??**, os endereços do armazém são formados a partir de uma combinação de três *dimensões*: Prateleira, Linha e Nível. Cada combinação dessas três propriedades é capaz de alocar um *pallet*, que por sua vez pode conter diversas unidades de um mesmo item.

Imagine um item de código 22B12, por exemplo, que representa um produto X. Este produto é disposto em um *pallet* com capacidade de armazenar 30 unidades do item 22B12. O armazém é composto por 26 prateleiras, sequenciadas de 'A' a 'Z'. Cada prateleira possui 2 linhas de profundidade e 3 níveis de altura. O *pallet* com o código 001 contém 30 unidade do item 22B12, e precisa ser alocado dentro armazém, para isso é utilizado um sistema de códigos envolvendo as 3 dimensões do armazém.

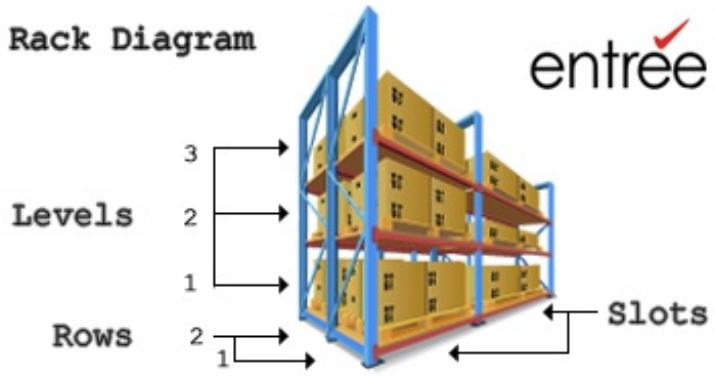


Figura 10 – Diagrama de dimensões do armazém (REFAZER)

Desta forta 60 unidades do item 22B12 estão dispostos em dois *pallets*(001, 002) existentes no armazém e cada um dos *pallets* será destinado a um endereço. O *pallet* 001 será alocado na terceira prateleira, no segundo nível e na primeira linha. Após a formação do endereço, o *pallet* 001 estará localizado no endereço C0101, prateleira C, linha 01 e nível 01.

4.1 Hipóteses

As hipóteses na diferença de desempenho entre as APIs derivaram dos fundamentos teóricos. Elas baseiam-se no entendimento de que os protocolos utilizados possibilitam a implementação de uma combinação de técnicas afim de afetar positivamento o desempenho da API. Portanto, ambas as implementações devem ter as mesmas propriedades, seguindo suas melhores práticas, modelos de maturidade e documentação.

As hipóteses deste trabalho são:

- O tamanho da resposta será menor utilizando GraphQL;
- O tempo de resposta será menor utilizando GraphQL;
- A utilização da CPU será menor utilizando REST;
- O consumo de memória será menor Utilizando REST;

Com o propósito de validar as hipóteses definidas, foram determinadas duas perguntas que envolvessem todas as entidades. Para cada pergunta existe apenas uma resposta correta e sua lógica é baseada em campos das estruturas de dados de retorno.

Questão 1: Qual item ocupa a maior quantidade de *pallets* alocados no armazém?

Questão 2: O item 22B12 está armazenado em quais endereços?

4.2 Cenários

4.3 Ferramentas utilizadas

A escolha das ferramentas a serem usadas na implementação foi uma das partes mais importantes no planejamento do estudo de caso. Foi necessário pensar em uma especificação que apresentasse uma curva rápida de aprendizagem, um fluxo de execução replicável e independente da plataforma, e uma documentação madura em relação a implementação das APIs.

As APIs foram escritas na linguagem de programação JavaScript, utilizando a especificação do ECMAScript 5. Foi escolhido esta linguagem devido ao fácil acesso a boas ferramentas para construção de serviços Web, como o Node.js. Node.js é uma plataforma para desenvolvimento de servidores Web baseada em rede utilizando JavaScript e o V8 JavaScript Engine, assim, com Node.js pode-se criar uma variedade de aplicações Web utilizando código em JavaScript (**??**).

4.3.1 Ferramentas servidor

Node.js dispõe de inúmeros recursos e ferramentas que possibilitam a construção de APIs. Mesmo assim, o ecosistema de bibliotecas no Node.js conta com ferramentas que simplificam ainda mais a construção de aplicações para servidores Web. Nesta sessão seão citadas algumas bibliotecas que foram utilizadas para desenvolver tanto a API REST quando a API GraphQL.

Express.JS

Express é um *framework* para Node.js extremamente flexível, que fornece um conjunto robusto de ferramentas para a construção de aplicações Web e Móveis. Conta com um robusto sistema de roteamento, facilitando o desenvolvimento de APIs. O Express fornece uma fina camada de abstração nas principais funcionalidades do Node.js, sem sobrepor seus recursos.

Nos protótipos desenvolvidos para executar o experimento, o Express atua como um *middleware* de gerenciando as rotas e delegando a responsabilidade de interpretação das requisições para os *Controllers* na API REST e para os *Resolvers* na API GraphQL.

O código **??** ilustra como os *Controllers* enviam as informações da requisição para o model, que é o responsável na API REST em executar as consultas no banco de dados. No exemplo abaixo, o trexo de código é o responsável por retornar um Item baseado no id recebido como parâmetro da requisição.

```
1 //item.controller.js
2 import Item from '../models/item.model';
3
4 /**
5  * Load item and append to req.
6  */
7 function load(req, res, next, id) {
8   Item.get(id)
9     .then((item) => {
10       req.item = item;
11       return next();
12     })
13     .finally(e => next(e));
14 }
```

Código 1 – Controller para carregar um item

Já o código **??** mostra como o Express.JS gerencia a requisição recebida via método GET, e delega a responsabilidade para o schema.js, onde está contida a lógica para a interpretação dos parâmetros da requisição, e retorna um objeto JSON com a devida resposta.

```
1 //server.js
2 import express from 'express';
3 let app = express();
4
5 import schema from './schema.js';
6
7 app.get('/', (req, res) => {
8     graphql(schema, req.query.query)
9     .then((result) => {
```

```
10          res.send(result);
11      });
12 });
```

Código 2 – Expess gerenciando rotas para GraphQL

MongoDB

MongoDB é um banco de dados *open source* que utilizada um modelo de dados orientado a objetos. Ele tem como característica conter todas as informações importantes em um único documento, possuir identificadores únicos universais (UUID), possibilitar a consulta de documentos através de métodos avançados de agrupamento e filtragem, também conhecido como *MapReducers*. Ao invés de usar tabelas e linhas, como os bancos de dados relacionais, o MongoDB usa uma arquitetura baseada em coleções e documentos.

Ferrametas clientes

Algumas ferramentas foram utlizadas para se comportaram como clientes das APIs construídas, e foram responsável por executar as consultas nas APIs. O foco deste trabalho não é em cima das aplicação clientes, entretanto é importante conhecê-las pois são elas que invocarão as consultas como também é através delas que algumas métricas serão extraídas. O software Postman será usado fundamentalmente para a execução das buscas nas APIs.

O Postman é uma cadeia completa de ferramentas para desenvolvedores de APIs. Ele é uma ferramenta elegante e flexível usada para construir softwares conectados via APIs, de forma rápida, fácil e precisa.

Ele funciona como uma emulador para execução de consultas em APIs. Com ele é possível executar buscas utilizando qualquer dos métodos do protocolo HTTP, possibilitando personalização tando do corpo quando do textup das requisições, se preciso. Junto com a resposta da consulta, o Postman traz também informações extremamente relevantes, como tempo que a API levou para responder e o tamanho em bytes contido na resposta. estas funcionalidades alidas com a opção de executar um número personalizável de iterações para cada requisição, será a base para avaliar o desempenho das APIs REST e GraphQL.

4.3.2 Ambiente

A configuração da máquina utilizada para o teste de desempenho local é descrita na Tabela **??**. As APIs REST e GraphQL foram construídas para responder às requisições recebidas, retornando respostas no formato JSON, a fim de gerar um fator de medição de desempenho da execução dos testes. Para a execução do experimento, apenas os softwares necessários estavam ativos. Portanto, ao executar as buscas, o servidor REST ou GraphQL

estará ativo, além do servidor de banco, as aplicações clientes, e as aplicações de coleta das métricas.

Item	**Descrição**
Marca/Modelo	Mac
Processador	Intel
Memória	X GB
Disco rígido	SSD 128
Quantidade núcleos	4

Tabela 1 – Configuração do Ambiente (–REVER–)

4.3.3 Detalhes implementação REST

O servidor REST consiste em um servidor HTTP escrito em Node.js que recebe as requisições HTTP recebidas. Dependendo do método e URL da requisição, roteia-o para o *controller* correspondente. O *controller* faz a consulta na base de dados do MongoDB. Ao realizar o gerenciamento das rotas, o servidor realiza os devidos registros de latência. Após a consulta, a resposta desejada é enviada de volta para o cliente. Esse fluxo acontece em qualquer cenário, independente do número de requisições desejadas.

No presente trabalho, como o objetivo é apenas medir o desempenho de consultas utilizando REST, todas as requisições utilizarão o método HTTP GET. Para as medições da API REST, a aplicação cliente enviará por exemplo, uma requisição para recuperar todos os itens cadastrados. Como pode ser observado da tabela **??**, é necessário executar uma busca do tipo /item, que será interpretada pelo servidor REST, identificando qual é a rota responsável por essa requisição. O servidor consultará a base de dados retornando uma resposta no formato JSON, com todos os itens cadastrados na API. Esse fluxo é ilustrado na figura **??**

4.3.4 Detalhes implementação GraphQL

O servidor GraphQL foi implementado também usando Node.js. A diferença em comparação com a implementação do servidor REST é que o servidor GraphQL envia todos os pedidos ao seu núcleo, ao invés de rotear as requisições recebidas para vários *controllers* diferentes. O GraphQL analisa a consulta e envia os parâmetros para os *resolvers* responsáveis, localizados nos *schemas*. *Resolvers* são funções definidas para todos os campos no *schema*, cada um retorna os dados para o campo específico. Estas funções são executadas quando os campos correspondentes são consultados e os resultados são retornados na resposta.

```
1 query RootQuery {
2   items {
```

URI	**Descrição**
/item	Consulta lista de itens
/item/:id	Consulta item pelo id
/pallet	Consulta lista de pallets
/pallet/:id	Consulta pallet pelo id
/address	Consulta lista de endereços
/address/:id	Consulta endereço pelo id
/slot	Consulta lista de prateleiras
/slot/:id	Consulta prateleira pelo id
/row	Consulta lista de linhas
/row/:id	Consulta linha pelo id
/level	Consulta lista de nível
/level/:id	Consulta nível pelo id

Tabela 2 – Servidor REST

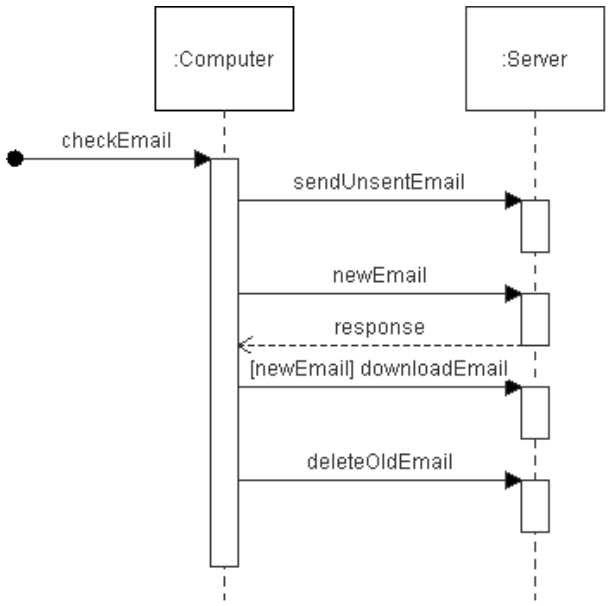


Figura 11 – UML sequence REST (REFAZER)

```
3         id
4         description
5       }
6 }
```

Código 3 – Consulta de itens

A consulta acima retorna como resposta lista a de todos os itens cadastrados na base de dados. Note que a resposta contém apenas os atributos que a consulta requisitou:

```
1 {
2       "data": [
3             {
4               "id": 22B12
```

```
5             "description": "Flat screen",
6         },
7         {
8           "id": 21C44
9             "description": "Computer screen",
10        },
11        {
12          "id": 43F12
13            "description": "Smartphone screen",
14        },
15    ]
16 }
```

Código 4 – Listagem dos itens (Validar)

Aqui tenho que explicar o diagrama de sequência para o GraphQL

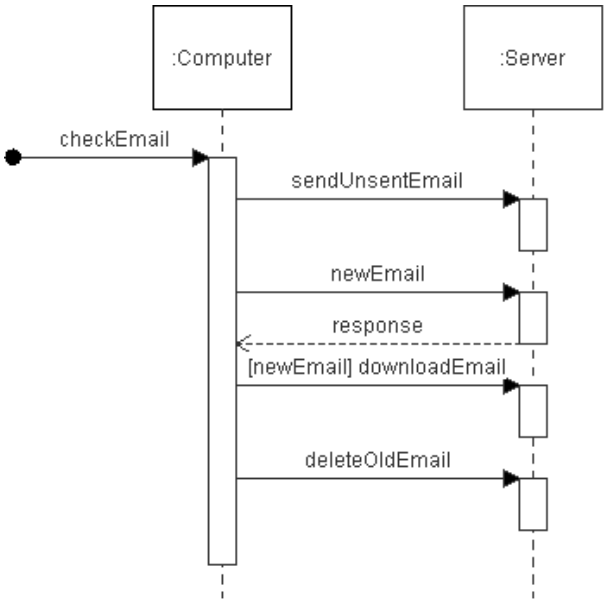


Figura 12 – UML sequence GraphQL (REFAZER)

4.4 Métricas

A fim de comparar as medidas de desempenho de APIs desenvolvidas em REST com APIs desenvolvidas em GraphQL, algumas métricas precisam ser definidas. O desempenho de cada API vai depender de sua implementação, porém, escolhendo as métricas corretas, o efeito da implementação pode ser reduzido. Focando em medições corretas, as diferenças relevantes das APIs podem ser destacadas e melhor ponderadas.

Para o presente trabalho, serão utilizadas quatro métricas diferentes: Utilização da CPU, Consumo de memória, Tempo de resposta e o Tamanho da resposta. É importante

ressaltar que cada métrica foi medida separadamente, para que *logs* e *outputs* pertinentes a uma métrica específica não interfira no resultado das demais.

Utilização da CPU

A utilização da CPU é uma medida do percentual de ciclos que representa uma porcentagem em que as unidades de processamento ficam dedicadas a executar um processo em particular. No contexto do presente trabalho, é a porcentagem de processamento dentro da CPU utilizada em cada requisição.

Esta métrica será extraída atraves do módulo *OS* presente no core do Node.Js. A função *os.loadavg()* trás como resultado uma medida da atividade do sistema, calculada pelo sistema operacional e expressa como um número fracionário. A média será a medida de comparação entre as duas abordagens. A seguinte fórmula será usada para calcular esta métrica:

$$S = mrs1 + mrz2.. + mrsn$$

Onde MRS é o tempo total para obtido em cada requisição necessária, e S representa a soma dos tempos, resultando do

Consumo de memória

O consumo de memória, junto com a utilização da CPU, é uma das medidas mais importantes pois são nesses pontos que observamos o verdadeiro custo por trás da escolha da ferramenta. O consumo de memória é a quantidade de em bytes utilizada pela API, e será medido através da soma da utilização de memória em cada consulta necessária para atender os cenários propostos. A seguinte fórmula será usada para calcular esta métrica:

$$S = mrs1 + mrz2.. + mrsn$$

Onde MRS é o tempo total para obtido em cada requisição necessária, e S representa a soma dos tempos, resultando do

Ferramenta X utilizada

Tempo de resposta

É o tempo que cada requisição levou para realizar a consulta. Esse tempo é calculado a partir do início da requisição, até o retorno da resposta completa. No caso da API REST, esta métrica será cumulativa, ou seja, a soma do tempo de todas as requisições necessárias. A seguinte fórmula será usada para calcular esta métrica:

$$S = mrs1 + mrz2.. + mrsn$$

Onde MRS é o tempo total para obtido em cada requisição necessária, e S representa a soma dos tempos, resultando do

Tamanho da resposta

O tamanho da resposta será calculada baseado no tamanho em *bytes* da resposta. Essa métrica será calcula a partir da média de 30 consultas. Novamente, para a API REST será usada a média das somas de todas as consultas necessárias para obter a resposta desejada. A seguinte fórmula será usada para calcular esta métrica:

$$S = mrs1 + mrz2.. + mrsn$$

Onde MRS é o tempo total para obtido em cada requisição necessária, e S representa a soma dos tempos, resultando do

Conclusão

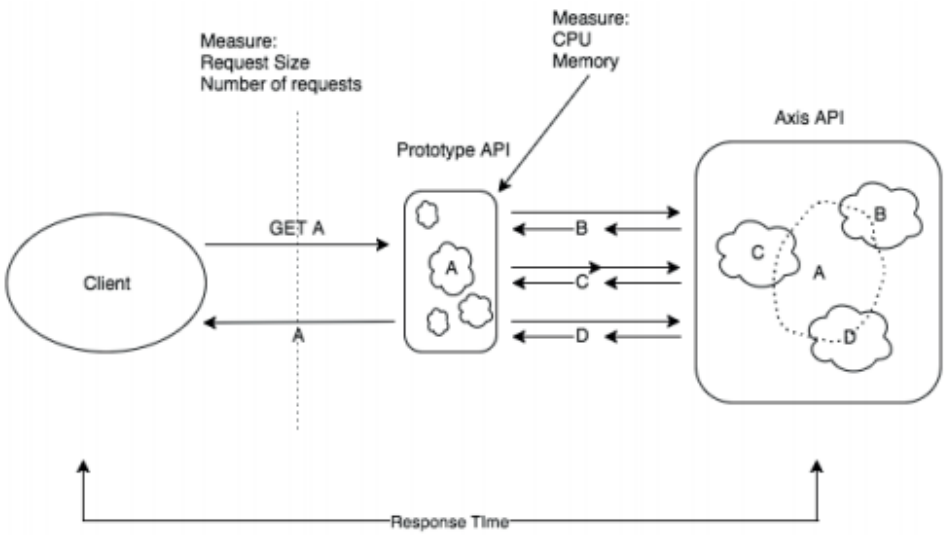


Figure 3.1: System model with the different measurements pointed out

Figura 13 – Arquitetura das APIs e diferentes pontos de medidas (REFAZER)

A figura **??** mostra como as métricas serão extraidas. A utlização do CPU e o consumo de memória serão medida utilizando ferramentas do próprio Node.js. Ambos serão medidos via *logs* no código fonte dos protótipos implementados. O tamanho da resposta e o tempo de resposta serão extraídos utilizando a ferramenta Postman, ao final de cada consulta.

5 Resultados

Neste capítulo, os resultados e a análise dos experimentos realizados durante o trabalho são reportados. Os resultados obtidos nos testes de validação apresentam a média dos valores coletados após diversas execuções dos cenários de forma sequencial.

Foram executadas 30 interações para cada requisição necessária a fim de obeter as respostas das questões levandas no capítulo anterior. Os dados necessários puderam ser buscados em uma única requisição à API GraphQL, enquanto foram precisos à API REST duas requisições para chegar a resposta da primeira questão, e vinte e oito para obter a resposta da segunda. Os detalhes podem ser constatados nas tabelas **??** e **??**.

Requisição	**Resultado**	**Número de requisições**
/pallets	Todos os pallets	1x
/items/:id	Detalhes do item mais presente	1x

Tabela 3 – Fluxo de dados para responder Questão 1

Requisição	**Resultado**	**Número de requisições**
/items	ID do item 22B12	1x
/items/:id	Detalhes do item 22B12	1x
/pallets	Pallets contendo item 22B12	1x
/pallets/:id	Detalhes do Pallet contendo o item 22B12	5x
/addresses/:id	Detalhes do Endereço contendo o item 22B12	5x
/levels/:id	Nível contendo o item 22B12	5x
/slots/:id	Prateleira contendo o item 22B12	5x
/rows/:id	Linha contendo o item 22B12	5x

Tabela 4 – Fluxo de dados para responder Questão 2

Além das 30 interações feitas para cada requisição, foi montado também três cenários para melhor analise dos resultados. Estes cenários possuem quantidade de registros diferentes para as entidades Item e Pallet, pois são estas que afetam de maneira mais significativa o quão eficiente será a resposta das APIs. Estes três cenários estão descritos na tabela **??** com suas respectivas quantidades.

5.1 Questão 1

Embora necessite de buscas mais simples, através da Questão 1 já é possível observar as diferenças em termos de desempenho entre a aplicação REST e a aplicação GraphQL. A Questão 1 busca o Item com a maior quantidade de pallets alocados no armazém, e para responde-la é necessário duas etapas. A primeira busca todos os pallets registrados

Recurso	**C1**	**C2**	**C3**
Item	1000	10000	30000
Pallet	1000	10000	30000
Address	156	156	156
Slot	26	26	26
Row	2	2	2
Level	3	3	3

Tabela 5 – Cenários analisados

no sistema, e após identificado qual o item mais comum presente nos *Pallets* registrados, a segunda etapa detalha este item, extraindo a descrição dele por exemplo.

5.1.1 Utilização da CPU

Ao analisarmos a utilização da CPU, ilustrada na figura **??**, percebemos que as APIs REST e GraphQL tem desempenho similares nos cenários C1 e C2. Por outro lado, as consultas para o cenário C3, mostram a API REST muito menos eficiente ao utilizar a CPU.

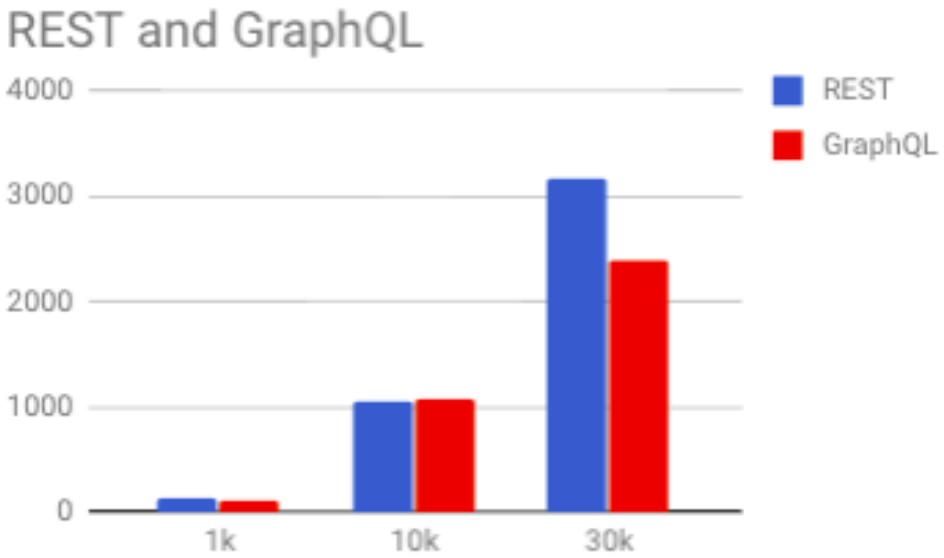


Figura 14 – Comparação da Utilização de CPU

As consultas de C1 exigiram 133.8 ms de CPU na API REST e apenas 111.10 ms na API GraphQL. Executando o cenário C2, a API REST foi processada em 1042.52 ms enquanto a API GraphQL levou 1065.45 ms, e é neste cenário que notamos a maior similaridade no tempo de utilização da CPU entre as APIs. No entanto, quando analisado os resultados de C3, nota-se uma grande desvantagem para API REST, que utilizou 3177.6 ms da CPU enquanto a API GraphQL usou apenas 2382.1 ms uma diferença de cerca de 25%.

5.1.2 Consumo de memória

Os dados do resultado da comparação do consumo de memória expõem que a API REST também se mostrou menos eficiente neste quesito em relação a API GraphQL. Os resultados podem ser obervados na figura **??**

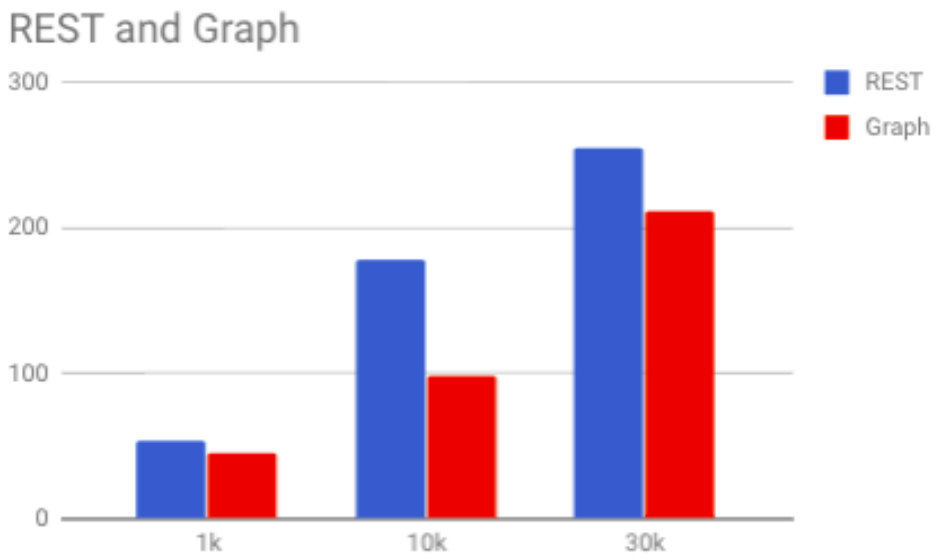


Figura 15 – Comparação do Consumo de Memória

Ao compararmos os resultados de C1 e C3, a API REST se mostrou cerca de 15% menos eficiente que a API GraphQL. O que se destaca é quando comparamos os resultados de C3, em que a API GraphQL consumiu 127.71 megabytes de memória e a API REST consumiu 178.01 megabytes, uma diferença de quase 30%.

5.1.3 Tempo de resposta

Como esperado, a API implementada com GraphQL realmente respondeu as consultas em um tempo menor do que a API REST. A figura **??** mostra a diferença do tempo de respota das APIs para executar as consultas da primeira questão.

Nas requisições de C1, a API REST teve como resultado um tempo de resposta de 147.23 ms, enquanto a API GraphQL respondeu a consulta em 115.63 ms, representando uma diferença de 21% por cento. Ao analizar as consultas de C2, a API REST respondeu as consultas em 1108.13 ms e a API GraphQL devolveu os resultados em 925.63 ms, uma diferença de 16% por cento. Por último, as consultas de C3 foram respondidas em 2261.1 ms na API REST e 1725.7 ms na API GraphQL, o que representa uma diferença de 23% por cento.

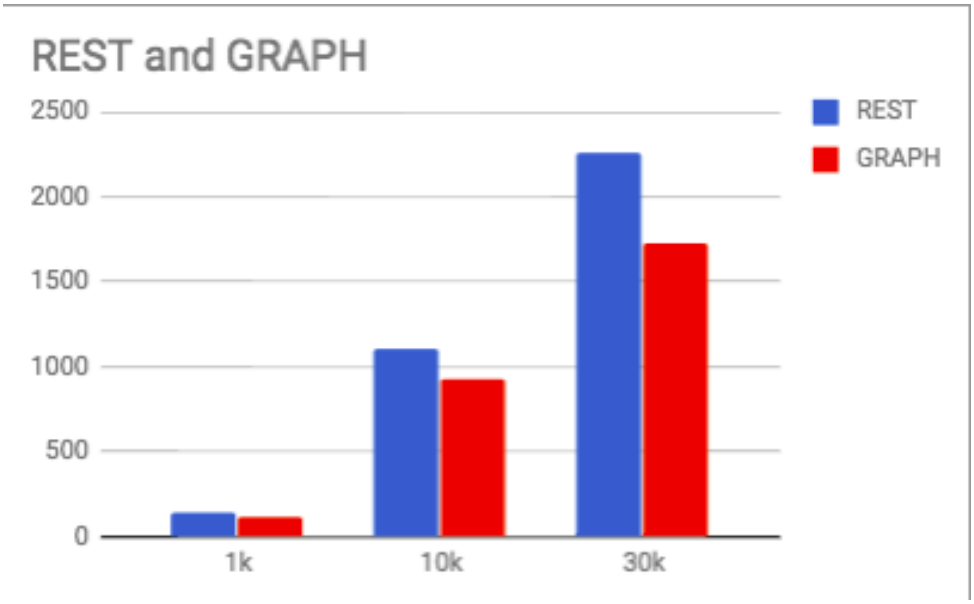


Figura 16 – Comparação do tempo de resposta

5.1.4 Tamanho da resposta

Outro resultado esperado era que o tamanho da resposta da API GraphQL fosse menor do que o tamanho da resposta da API REST. Essa hipótese se confirmou como pode ser visto da figura **??**.

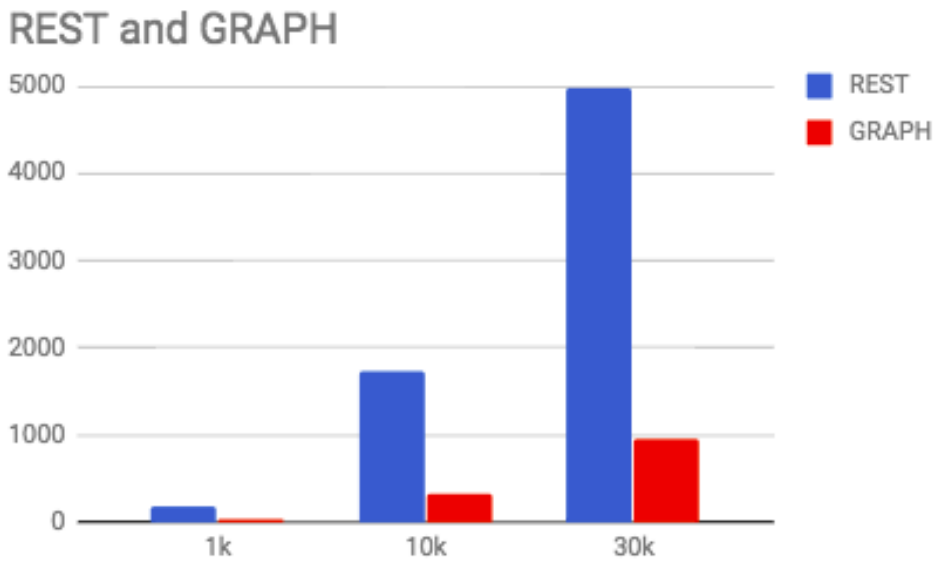


Figura 17 – Comparação do tamanho de resposta

A API REST respondeu as requisições da Questão 1 com um tamanho de resposta de 174.04 Kb, 1740.17 Kb e 4980 Kb para C1, C2, C3 respectivamente. Da mesma maneira, a API GraphQL teve como resultado respostas com 31.68 Kb, 322.53Kb e 967.35 Kb. Comparando os três cenários é constatado uma diferença constante de cerca de 80% entre

a API REST e a API GraphQL.

5.2 Questão 2

Para as repostas da questão 2, as consultas foram mais complexas na API GraphQL, e mais numerosas na API REST.

5.2.1 Utilização da CPU

Analisando os resultados da utilização de CPU, podemos concluir que a API GraphQL utiliza este recurso de uma maneira mais eficiente do que a API REST. A figura **??** ilustra os resultados obtidos.

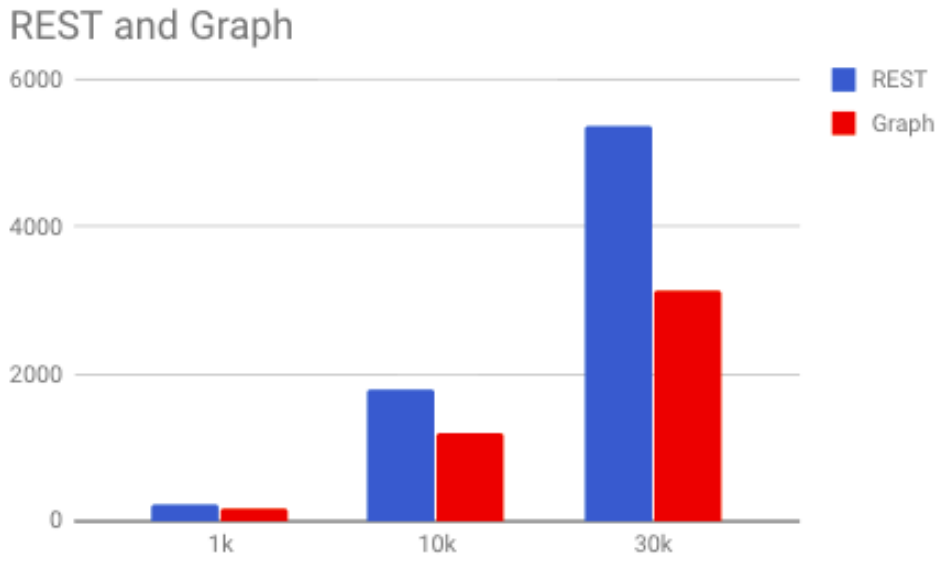


Figura 18 – Comparação da Utilização da CPU - Q2

Aqui a diferença nos resultados obtidos é notada com mais clareza a medida em que aumentamos o número de registros. Analizando C1, a API REST exiigu 244.41 ms da CPU para ser processada, ao mesmo tempo que a API GraphQL exigiu somente 178.22 ms. Nas consultas de C2, a API REST demandou 1787.74 ms para ser processada, e a API GraphQL demandou 1199.53 ms. A maior diferença encontra-se em C3, onde a API REST levou 5383.40 ms para ser completamente processada pela CPU e a API GraphQL levou apenas 3132.98 ms. No cenário C3 observamos uma diferença de mais de 40% entre o desempenho das APIs.

5.2.2 Consumo de memória

Os dados referentes ao consumo de memória estão ilustrados na figura **??**, e apontam que a API REST faz uso menos eficiente deste recurso, comparando com o consumo de

memória da API GraphQL. Foi analisando esta métrica que se encontrou o único cenário em que a API REST se mostrou mais eficiente que a API GraphQL.

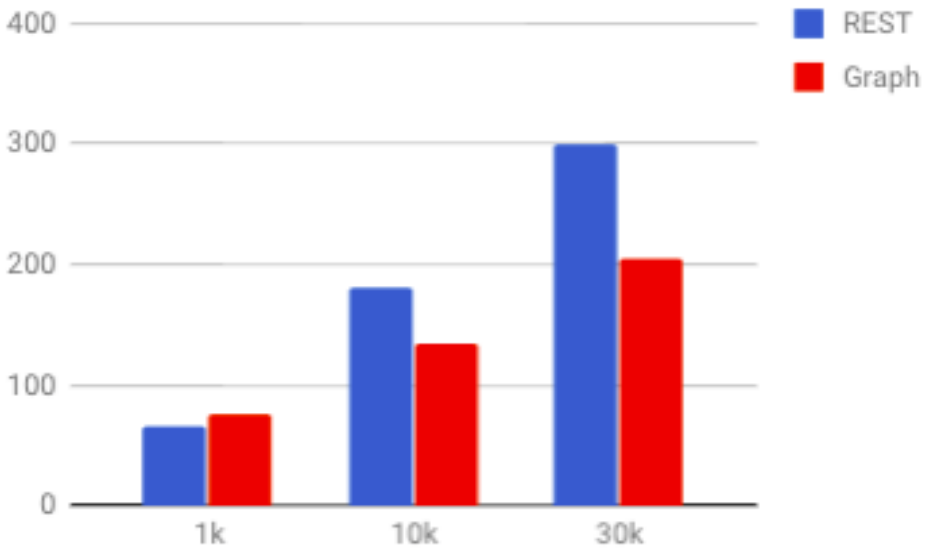


Figura 19 – Comparação do Consumo de memória Q2

O consumo de memória não se mostrou tão diferente comparando as APIs nos cenários C1 e C2. Nas requisições de C1, a API REST demonstra ser mais eficiente, mesmo que a diferença seja de apenas 9.88 megabytes (ou 12%), em relação a API GraphQL. A API GraphQL consumiu 76.19 mb de memória, contra 66.27 consumidos pela API REST. Essa melhor eficiencia já não é mais identificada em C2, onde a API REST consumiu 181.02 mb de meória, e a API GraphQL consumiu 26% mais, totalizando 133.52 mb. Nas consultas de C3, observamos uma diferença relevante no consumo de mémoria, com a API REST consumindo 300.30 mb de memória enquanto a API GraphQL consumiu 206.02, uma impressionante diferença de quase 100 mb ou 30%.

5.2.3 Tempo de resposta

Como mostra a figura **??**, é possivel identificar com clareza que a API REST leva um tempo consideravelmente maior para responder todas as requisições comparando com o tempo levado pelas consultas na API GraphQL.

Considerando as consultas de C1, a API REST teve como resultado um tempo de resposta de 254.56 ms, enquanto a API GraphQL respondeu a consulta em 148.5 ms, representando uma diferença de pouco mais de 40%. Quando analisado as requisições de C2, a API REST respondeu as consultas em 2072.03 ms e a API GraphQL devolveu o resultado em 1201.3 ms, uma diferença de 42%, muito semelhante ao C1. Por último, as consultas de C3 foram respondidas em 4770.2 ms na API REST e 2291.1 ms na API GraphQL, o que representa uma diferença de pouco mais de 50%.

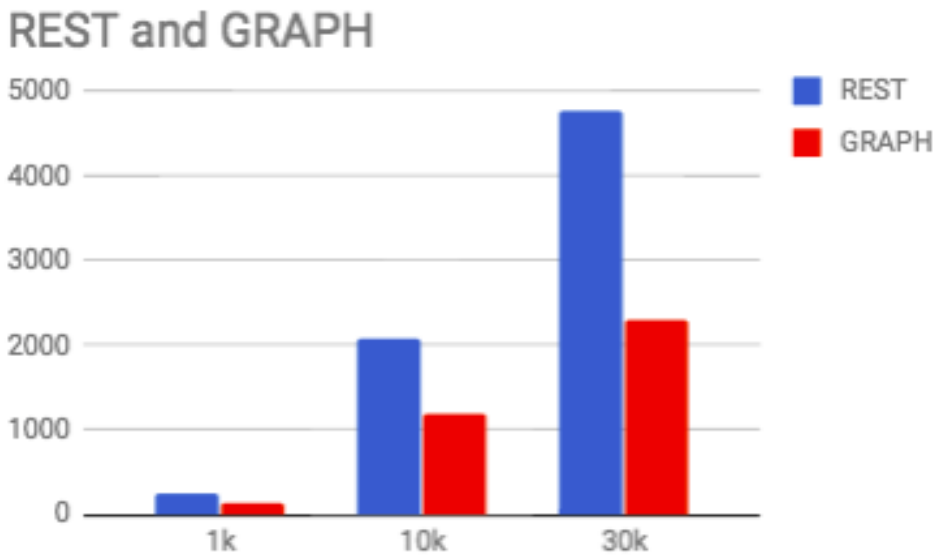


Figura 20 – Comparação do tempo de resposta Q@

5.2.4 Tamanho da resposta

A API GraphQL também mostrou-se mais vantajosa em termos de tamanho da resposta na Questão 2. A figura **??** mostra a comparação entre os protótipos quando comparados esta métrica.

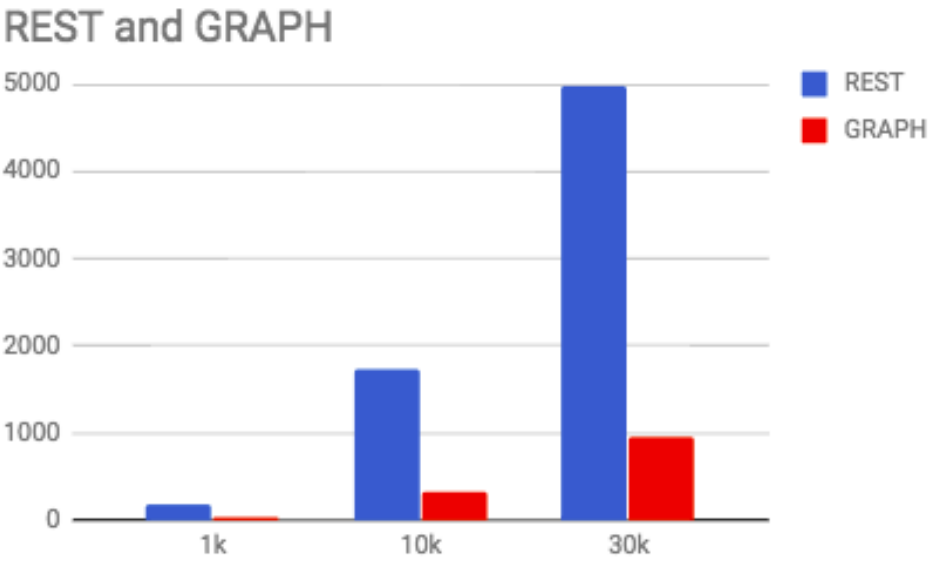


Figura 21 – Comparação do tamanho da resposta

A API REST respondeu as requisições da questão 1 com um tamanho de resposta de 259.53 Kb, 2522.17 Kb e 7221 Kb para C1, C2 e C3 respectivamente. Da mesma maneira, a API GraphQL teve como resultado respostas com 101.73 Kb, 1005.23 Kb e 2850 Kb. Como ocorrido da Questão 1, a diferença de desempenho entre as APIs se manteve

em torno de 60% nos três cenários, deixando claro a melhor eficiencia da API GraphQL neste quesito.

6 Conclusão

Sed consequat tellus et tortor. Ut tempor laoreet quam. Nullam id wisi a libero tristique semper. Nullam nisl massa, rutrum ut, egestas semper, mollis id, leo. Nulla ac massa eu risus blandit mattis. Mauris ut nunc. In hac habitasse platea dictumst. Aliquam eget tortor. Quisque dapibus pede in erat. Nunc enim. In dui nulla, commodo at, consectetuer nec, malesuada nec, elit. Aliquam ornare tellus eu urna. Sed nec metus. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas.

Phasellus id magna. Duis malesuada interdum arcu. Integer metus. Morbi pulvinar pellentesque mi. Suspendisse sed est eu magna molestie egestas. Quisque mi lorem, pulvinar eget, egestas quis, luctus at, ante. Proin auctor vehicula purus. Fusce ac nisl aliquam ante hendrerit pellentesque. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Morbi wisi. Etiam arcu mauris, facilisis sed, eleifend non, nonummy ut, pede. Cras ut lacus tempor metus mollis placerat. Vivamus eu tortor vel metus interdum malesuada.

Sed eleifend, eros sit amet faucibus elementum, urna sapien consectetuer mauris, quis egestas leo justo non risus. Morbi non felis ac libero vulputate fringilla. Mauris libero eros, lacinia non, sodales quis, dapibus porttitor, pede. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Morbi dapibus mauris condimentum nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Etiam sit amet erat. Nulla varius. Etiam tincidunt dui vitae turpis. Donec leo. Morbi vulputate convallis est. Integer aliquet. Pellentesque aliquet sodales urna.