UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA

Mateus Maso

ESTUDO SOBRE O USO DA LINGUAGEM GRAPHQL NA COMPOSIÇÃO DE DADOS ATRAVÉS DE SERVIÇOS BASEADOS EM JSON

Florianópolis, SC

Prof. Dr. Frank Augusto Siqueira

Mateus Maso

ESTUDO SOBRE O USO DA LINGUAGEM GRAPHQL NA COMPOSIÇÃO DE DADOS ATRAVÉS DE SERVIÇOS BASEADOS EM JSON

Trabalho de Conclusão de Curso submetido ao Programa de Graduação em Sistemas de Informação para a obtenção do Grau de Bacharel em Sistemas de Informação.

Florianópolis, SC

Prof. Dr. Frank Augusto Siqueira

Este trabalho é dedicado a minha família.

AGRADECIMENTOS

Inserir os agradecimentos aos colaboradores à execução do trabalho.

Time has told me not to ask for more,
someday our ocean will find its shore
(Nick Drake)

RESUMO

O texto do resumo deve ser digitado, em um único bloco, sem espaço de parágrafo. O resumo deve ser significativo, composto de uma sequência de frases concisas, afirmativas e não de uma enumeração de tópicos. Não deve conter citações. Deve usar o verbo na voz passiva. Abaixo do resumo, deve-se informar as palavras-chave (palavras ou expressões significativas retiradas do texto) ou, termos retirados de thesaurus da área.

Palavras-chave: GraphQL, REST, JSON Schema.

ABSTRACT

Resumo traduzido para outros idiomas, neste caso, inglês. Segue o formato do resumo feito na língua vernácula. As palavras-chave traduzidas, versão em língua estrangeira, são colocadas abaixo do texto precedidas pela expressão "Keywords", separadas por ponto.

Keywords: GraphQL, REST, JSON Schema.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

SUMÁRIO

1 FUNDAMENTOS

Neste capítulo são estabelecidos os principais conceitos utilizados ao longo do projeto. Será feito uma breve abordagem sobre os fundamentos de representação de dados; seguido pelo formato de serialização no qual será estudado e, por fim, tipos de serviços e tecnologias utilizadas para desenvolver a ferramenta.

1.1 SERIALIZAÇÃO DE DADOS

Na ciência da computação, serialização de dados é um processo de tradução usado para converter estruturas de dados[1] em formatos que possam ser armazenados, transmitidos e reconstruídos por um mesmo ou outro ambiente computacional. (**??**)

Dados serializados normalmente vivem mais tempo que suas aplicações de origem e, ao ser armazenado em disco ou transmitido pela rede, são representados de modo diferente que sua estrutura em memória. Para se ler dados serializados em memória é preciso realizar o processo inverso, também chamado de deserialização, onde estes passam a ser representados por estruturas da linguagem de execução. (**??**)

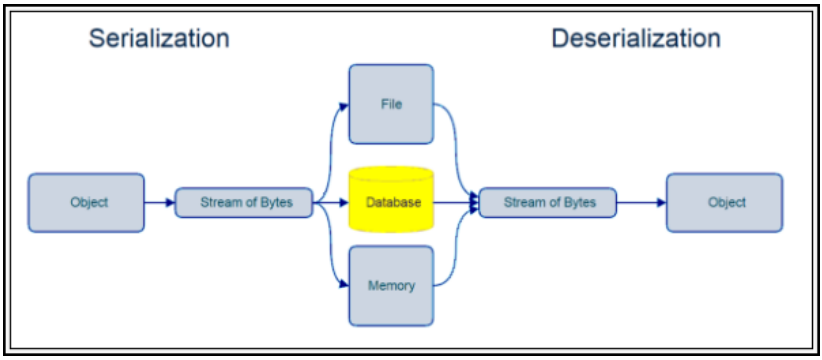

Figura 1 – Processo de serialização e deserialização

[1] Uma estrutura de dados é uma forma abstrata de representar e organizar dados. Seu objetivo é ajudar a reduzir complexidade, podendo armazenar dados de diferentes tipos, como números, strings ou até mesmo outras estruturas de dados.

Este processo, embora demande tempo, permitiu que aplicações fizessem o consumo de informações de forma distribuída, contribuindo com o aumento do volume de dados que circulam pela internet. Além disso, fez-se necessário a seleção adequada de formatos de serialização cuja estrutura não prejudique o desempenho de aplicações na busca por dados. (**??**)

Segundo a Cisco Systems[2], no ano de 2015, houve um aumento de 21% no volume de tráfego de dados registrados apenas por seus aparelhos. Sendo a categoria Web, Email e Data responsável por representar aproximadamente 7,558 petabytes de dados transmitidos por seus clientes durante um mês. (**??**)

Para suprir esta alta demanda, diversos formatos de serialização foram introduzidos para melhor atender os problemas de desempenho experienciados por serviços que disponibilizam dados serializados. Dentre eles, tempo de serialização e deserialização, tamanho de transferência, flexibilidade de uso, facilidade de leitura, automação, suporte para linguagem, entre outros. Contudo, cada formato possui seus prós e contras. (**??**)

Formato	Especificação	Codificação	Human-Readable	Esquema/IDL
XML	Padronizada	Textual	Sim	Sim
JSON	Padronizada	Textual	Sim	Parcial
YAML	Padronizada	Textual	Sim	Parcial
Avro	Padronizada	Binário	Não	Sim
Protocol Buffers	Padronizada	Binário	Parcial	Sim
Thrift	Não Padronizada	Binário	Parcial	Sim

Tabela 1 – Comparação de formatos de serialização

Para melhor entender o que define cada formato, será feito uma abordagem sobre as categorias de classificação usadas para estudar o comportamento dos formatos existentes hoje.

1.1.1 Especificação

Um formato pode ter sua especificação classificada como padronizada ou não padronizada. Uma especificação padronizada é regida por requisitos que auxiliam na reprodutibilidade do processo em outras linguagens para maximização da compatibilidade e minimização de erros. Ao contrário, não é garantido que sua implementação estejam

[2] Empresa de sistemas de rede.

seguindo os padrões e poderá ser considerado como não padronizada. (**??**)

1.1.2 Codificação

Codificação é o processo de sequenciamento de caracteres usado na redução da transmissão e armazenamento de dados. É possível classificar em dois tipos: textuais e binários. Formatos baseados em texto não são codificados para seu fácil acesso e podem ser lidos diretamente através de editores de texto. Já um formato binário faz o uso intensivo da codificação e decodificação para salvar espaço. (**??**)

1.1.3 Human-Readable

Ao representar estruturas de dados em formatos de serialização para que máquinas possam fazer a leitura, não é garantido, no entanto, que esta representação também seja legível por seres humanos.

Para que um formato seja human-readable, além de máquinas, pessoas precisam conseguir ler os dados serializados fora de contexto e ainda poder dizer qual dado está sendo representado. Para desenvolvedores, este detalhe é essencial no processo de debugging[3]. Apesar do processo de serialização de dados não prever a ideia de escrita manual, se um formato é legível por pessoas então também é possível ser descrito por elas. Em geral, a maioria dos formatos baseados em texto são human-readable, enquanto os formatos binários não são. (**??**)

Nota-se que a leitura de um formato é diferente de seu entendimento fora de contexto, uma vez que nem todos os formatos possuem maneiras de descrever metadados em suas estruturas. Por exemplo, JSON é um formato baseado em texto e tem como objetivo a facilidade de uso e legibilidade por desenvolvedores. Contudo, nem sempre é possível identificar o que está sendo representado em suas estruturas. Através de extensões como JSON-LD e JSON Schema, é possível descrever meta informações para melhor entender o que está sendo representado sem perder a estrutura original do formato JSON.

[3] Depuração é o processo de encontrar e reduzir defeitos num aplicativo de software ou mesmo em hardware.

1.1.4 Esquema/IDL

Com objetivo de entender o que está sendo representado, alguns formatos disponibilizam na sua especificação maneiras de descrever metadados para seus dados serializados. Esta categoria é importante principalmente para que máquinas consigam inferir quais estruturas estão sendo lidas e, assim, tomar decisões de forma autônoma.

Um formato descritivo normalmente disponibiliza estruturas como esquemas IDL[4] para descrição da própria representação. À medida que estas descrições são incorporadas dentro da mesma representação, é possível classificar estes formatos como sendo auto-descritivos. (**??**)

1.2 JSON

JSON ou Javascript Object Notation é um formato de serialização de dados human-readable baseado em texto com especificação padronizada e parcialmente descritivo. Foi desenvolvido por Douglas Crockford com o objetivo de representar dados em uma maneira simples, leve e flexível através da redução na sobrecarga de marcações comparado ao formato XML.

Por ter se adaptado bem no ambiente de aplicacoes distribuidas, este formato acabou sendo amplamento utilizado por empresas como principal forma de represetação de dados serializados em seus serviços. A figura 2 mostra claramente a preferência do formato JSON por desenvolvedores ao criar novas APIs. (**??**)

[4] Linguagem de descrição utilizada para descrever a interface dos componentes de um software.

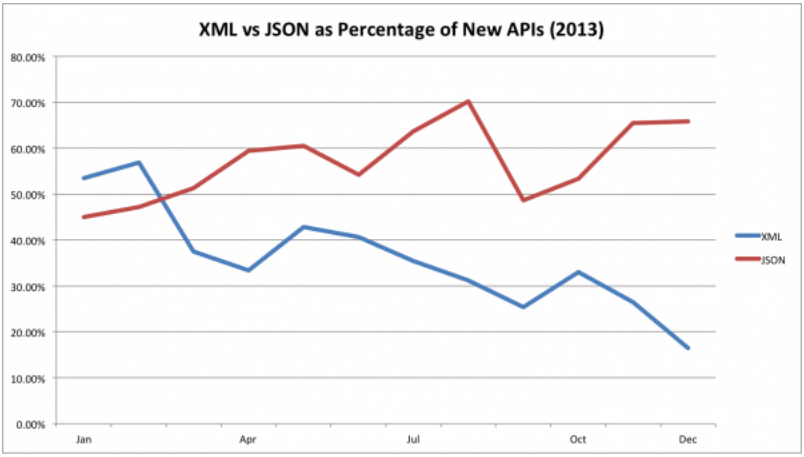

Figura 2 – Porcentagem de novas APIs em XML e JSON

Na sua essência, JSON foi construído com base em 4 tipos primitivos de dados e outros 2 para composição. Cada tipo possui seu respectivo correspondente na maioria das linguagens de programação, embora possam ser identificados por nomes diferentes. (**??**)

Tipo	Exemplo de Valor
Object	`{"key1": "value1", "key2": "value2"}`
Array	`["first", "second", "third"]`
Number	`1, -1, 2.9999`
String	`"This is a string"`
Boolean	`true, false`
Null	`null`

Tabela 2 – Exemplo de tipos de valores em JSON

Através da composição de listas, objetos e tipos primitivos, é possível representar complexas estruturas de dados que aplicações possam vir a serializar. No entanto, não existe um único padrão de representação em JSON, uma vez que dado uma estrutura para serializar, é possível representá-lo de inúmeras maneiras. (**??**)

Por exemplo, a seguir estão duas formas diferentes de representação em JSON para os mesmo dados de uma entidade "pessoa":

```
{
  "nome": "Mateus Maso",
  "aniversario": "25 de março de 1992",
  "cidade": "Florianópolis, SC, Brasil"
}
```

Figura 3 – Primeiro exemplo de representação JSON

```
{
  "nome": "Mateus",
  "sobrenome": "Maso",
  "nascimento": "25-03-1992",
  "cidade": {
    "nome": "Florianópolis",
    "estado": "SC",
    "pais": "Brasil"
  }
}
```

Figura 4 – Segundo exemplo de representação JSON

Ambas representações são válidas, apesar da figura 4 estar representando dados em uma estrutura mais formal que a outra. No entanto, por ser um formato não descritivo, a responsabilidade de entender o que está sendo representado em JSON vai depender da análise crítica ou conhecimento prévio dos desenvolvedores. Já uma máquina, sem conhecer o contexto, não saberia como interpretar os dados de forma correta. (**??**)

Para isso, será abordado em seguida um dos formatos de descrição existentes hoje para descrever estruturas JSON utilizados no projeto.

1.2.1 JSON Schema

JSON Schema é uma linguagem de definição projetada para descrever estruturas de dados JSON por meio de esquemas. Foi proposta em 2009 por Kris Zyp com objetivo de fornecer um contrato para que aplicações soubessem como trabalhar e interagir com certas estruturas de dados. Por meio deste, é possível prever representações e assim realizar operações de validação, documentação, navegação de hyperlink e controle de iteração.

Por ser uma linguagem de simples uso, para representar um esquema basta construir um objeto JSON utilizando um subconjunto válido de chaves especias descritas pela linguagem. No entanto, funcionalidades como descrição de estruturas multimídia[5] e a navegação de dados através de hyperlinks são apenas disponibilizadas no formato JSON Hyper-Schema, uma variação da linguagem de especificação. (**??**)

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "title": "Pessoa",
  "description": "Uma pessoa",
  "type": "object",
  "required": ["nome", "aniversario"],
  "properties": {
    "nome": {
      "type": "string"
    },
    "aniversario": {
      "type": "string"
    },
    "cidade": {
      "type": "string"
    }
  }
}
```

Figura 5 – Exemplo de JSON Schema para Figura 3

[5] Imagens, videos, audio digital.

```
{
  "$schema": "http://json-schema.org/draft-04/hyper-schema#",
  "title": "Pessoa",
  "description": "Uma pessoa",
  "type": "object",
  "required": ["nome", "aniversario"],
  "properties": {
    "nome": {
      "type": "string"
    },
    "aniversario": {
      "type": "string"
    },
    "cidade": {
      "type": "string"
    },
    "foto": {
      "media": {
        "binaryEncoding": "base64",
        "type": "image/png"
      }
    }
  },
  "links": [
    {
      "rel": "self",
      "href": "/{id}.json"
    },
    {
      "rel": "foto",
      "href": "/{id}.png",
      "mediaType": "image/png"
    }
  ]
}
```

Figura 6 – Exemplo de JSON Hyper-Schema para Figura 3

Com base nos esquemas especificados pelas figuras 5 e 6, ambos

asseguram que, dado uma estrutura JSON, para que esta seja reconhecido como uma entidade "pessoa" deve conter ao menos 2 propriedades descritas pela chave "required", além de seus valores estarem de acordo com o tipo descrito dentro da chave "properties". Já na figura 6, além das estruturas definidas pela figura 5, é descrito uma propriedade multimídia do tipo "foto" e, na chave "links", rotas para acesso e navegação de recursos em propriedades relacionadas.

Para validar objetos complexos em JSON, propriedades podem ser aninhadas e também representadas por subesquemas. Contudo, para evitar duplicação de estruturas, a linguagem oferece um poderoso sistema de composição e referenciamento de definições através da chave "$ref" e "definitions" vista a seguir na figura 7. (**??**)

```
{
  "$schema": "http://json-schema.org/draft-04/hyper-schema#",
  "title": "Pessoa",
  "description": "Uma pessoa",
  "type": "object",
  "required": ["nome", "aniversario"],
  "properties": {
    "nome": {
      "type": "string"
    },
    "aniversario": {
      "type": "string"
    },
    "cidade": {
      "type": "string"
    },
    "foto": {
      "media": {
        "binaryEncoding": "base64",
        "type": "image/png"
      }
    }
  },
  "links": [
    {
      "rel": "self",
      "href": "/{id}.json"
    },
    {
      "rel": "foto",
      "href": "/{id}.png",
      "mediaType": "image/png"
    }
  ]
}
```

Figura 7 – Exemplo de JSON Hyper-Schema para Figura 4 usando referenciamento

Em casos onde a complexidade de um esquema começa a crescer, é comum a definição de subesquemas através da chave “definitions”. Desta forma, podem ser referenciados permitindo o reuso de estruturas dentro de um esquema. Vale lembrar que a chave “definitions” é apenas um mecanismo simples e util para definir esquemas em um lugar comum, entretanto, não sugerem que estas propriedades sejam validadas em um objeto ao menos que referenciadas em outras estruturas do esquema. (**??**)

Como boa prática, é recomendado (mas não necessário) o uso da chave especial “$schema” para determinar quando uma estrutura JSON está sendo representada em forma de esquema. A seguir, será descrito algumas das chaves especiais usadas para descrever objetos em esquemas. (**??**)

Chave	Descrição	Exemplo de Valores
$schema	Identificador de versão	`http://json-schema.org/draft-04/schema##`
title	Nome da estrutura	`"Uma pessoa"`
description	Propósito da estrutura	`"Dados básicos de uma pessoa"`
required	Lista de prop. com presença obrigatória	`["nome", "sobrenome"]`
properties	Prop. usadas para validar uma estrutura	`{"nome": {"type": "string"}}`
definitions	Prop. e subesquemas para referência	`{"nome": {"type": "string"}}`
type	Tipo de dado	`"string", "number", "object"`
...	...	...
links	Lista de Link Description Objects	`[{"rel": "self", "href": "/{id}"}]`

Tabela 3 – Subconjunto de chaves especiais JSON Schema

De certa forma, JSON Schema continua sendo uma das únicas tentativas sérias de propor uma linguagem de definição em esquemas para estruturas JSON e, lentamente, está sendo estabelecida como padrão de uso para especificação JSON. Contudo, sua definição ainda está longe de ser um padrão universal, mas já há um número crescente de aplicações que suportam JSON Schema e uma quantidade significativa de ferramentas que permitem a validação de JSON Schemas. (**??**)

Vale lembrar também que, segundo Leach, com o recente surgimento de grandes formatos de descrição de APIs ao longo dos últimos anos como Swagger, Blueprint e RAML. JSON Schema tem-se tornado uma ótima opção de tecnologia para descrever API’s uma vez que utiliza da popularidade do formato JSON uma forma simples de descrever e disponibilizar esquemas pela web. Além disso, são extremamente nesses cenários, onde pode-se usar para evitar o recebimento de chamadas de APIs mal formadas que podem afetar o motor interno da aplicação. (**??**) (**??**)

Nos próximos capítulos vamos entrar mais em detalhe sobre tipos de serviços (API web) e de que forma é possível descrever APIs para o

leitura por máquinas.

1.3 REST

REST ou Representational State Transfer é um estilo arquitetural usado para a comunicação de sistemas distribuídos através do protocolo HTTP. Foi introduzido por Roy Fielding em 2000 com o objetivo de oferecer às aplicações web um modelo de interface de acesso baseada em recursos. Além disso, descreve 6 tipos de restrições que serviços deveriam aplicar para ganho de performance, escalabilidade, simplicidade, modificabilidade, visibilidade, portabilidade e confiabilidade.

Vale lembrar que, por ter causado grande repercussão após sua publicação. O termo REST, segundo Richardson, acabou sofrendo diversas interpretações durante o tempo e, sua descrição representada de formas não originalmente propostas por Fielding. Alguns descrevem que serviços que violam essas restrições não podem ser considerados RESTful. Para Wildermuth, apesar de reconhecer as vantagens de cada restrição, serviços web devem usá-los de forma pragmatica. (**??**) (**??**)

Apesar de ser introduzido num meio acadêmico, REST se adaptou bem em arquiteturas orientada a serviços. Segundo Pautasso, a eliminação da complexidade das soluções Web Services indicam que REST como uma solução aplicável para resolver problemas de integraçao de aplicaçoes empresariais e simplificar o encadeamento necessario para construir SOAs. A seguir, REST como adoção majoritária em relação a outros estilos para arquitetura de APIs. (**??**)

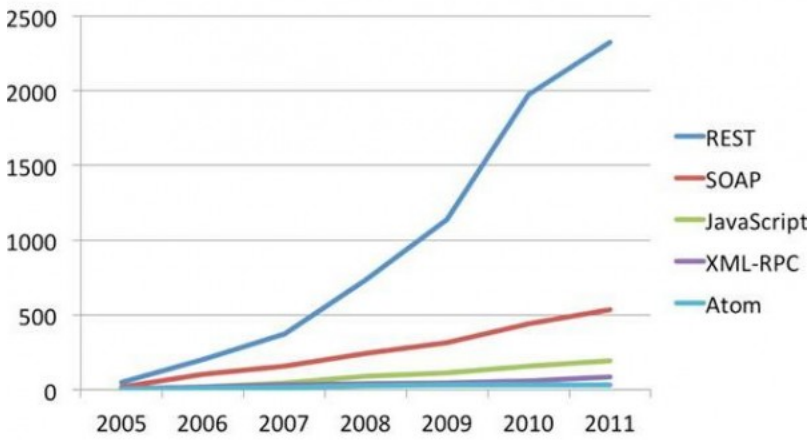

Figura 8 – Distribuição de estilos e protocolos para APIs

Sua maior vantagem de uso é por ser facilmente acessada por clientes web, mobile apps, dispositivos IoT. Isso permite que organizações construam clientes sem se preocupar com suporte à plataformas. Contudo, ao contrário de outros estilos de arquitetura, REST não sugere a criação de documentos de especificação, pois incentiva a escrita de respostas em hypertexo para navegação. Para Knupp, a idéia de documentação através de respostas autodescritivas dificulta a legibilidade, além de criar complexidade e informações adicionais. Ao invés, incentiva o uso de outras ferramentas de documentação e descrição de respostas. (**??**)

A seguir veremos sobre as restrições de arquitetura propostas pelo modelo REST, o que é uma API RESTful e seus níveis, além das atuais formas de documentação de APIs REST com suporte a leitura de máquinas.

1.3.1 Restrições de Arquitetura

Esta seção fornece uma visão geral sobre as restrições propostas por Fielding para a implementação em elementos dentro de uma arquitetura de serviço web. Além disso, será examinado o impacto de cada restrição em sistemas distribuídos em hypermedia.

1.3.1.1 Cliente-Servidor

É a primeira restrição para uma aplicação REST. Neste modelo, não existe conexão entre cliente e servidor, mas sim a espera do servidor por pedidos de clientes, executando solicitações e devolvendo uma resposta. Seu objetivo é separar arquitetura e responsabilidades dos ambientes cliente e servidor. Assim, o cliente (consumidor do serviço) não se preocupa com tarefas como a comunicação de banco de dados, gerenciamento de cache, etc. O inverso também é verdadeiro, onde o servidor (prestador de serviços) não está preocupado com as tarefas do cliente como interface, experiência do usuário etc. Assim, permitindo a evolução independente das duas arquiteturas desde que a interface de comunicacao entre os dois nao seja alterada. (**??**)

1.3.1.2 Sem Estado

Como HTTP é um protocolo sem conexão (onde não há nenhuma garantia sobre qual servidor será processado e quanto tempo irá levar) cada requisição deve conter todas as informações necessarias para que um servidor entenda o que um cliente está executando. Ou seja, para ser stateless, um servidor não pode guardar informações de estado do cliente, como sessões por exemplo. (**??**)

Esta restrição ajuda na viabilidade, confiabilidade e escalabilidade de sistemas distribuídos. Garantem que chamadas da API não estejam vinculadas a um determinado servidor. Contudo, com base no número da diversidade de clientes, ao manter um servidor sem estado é possível perder controle no tamanho de resposta, o que pode ser um fator crucial para aplicações que dependem disso. (**??**)

1.3.1.3 Interface Uniforme

Em essência, Fielding propõe que aplicações façam o uso de verbos HTTP (POST, GET, PUT, DELETE) e identificadores uniforme de recursos (URI) para mapear operações em ambientes distribuidos. Através de pequenas regras de acesso pelo cliente, é possível modificar/refatorar o servidor de maneira a minimizar riscos de acoplamento. Esta idéia, trouxe um pensamento diferente ao modelo RPC, cuja API dá ênfase a um maior número de operações. (**??**)

- Identificação de Recursos: Cada recurso deve ter uma URI coesa

e especifica para ser disponibilizado
- Representação de Recurso: Formato de representação no qual um recurso vai ser retornado para um cliente. (Exemplo: HTML, XML, JSON, TXT)
- Resposta Auto-explicativa: Uso de metadados na requisição e resposta. (Exemplo: código de resposta HTTP, Host, Content-Type)
- HATEAOS[6] - Retornar na resposta hyperlinks para que o cliente saiba navegar em busca de mais recursos relacionados.

1.3.1.4 Separação em Camadas

Um dos principios desta restrição está em evitar que clientes façam diretamente requisição para o servidor sem antes passar por um intermediario, como um load balancer ou alguma alguma outra máquina que faça ponte entre servidores. Assegurando que clientes apenas se preocupem com a comunicação, deixando para que intermediários lidem com a distribuição de requisições. (**??**)

[6] Hypermedia as the Engine of Application State.

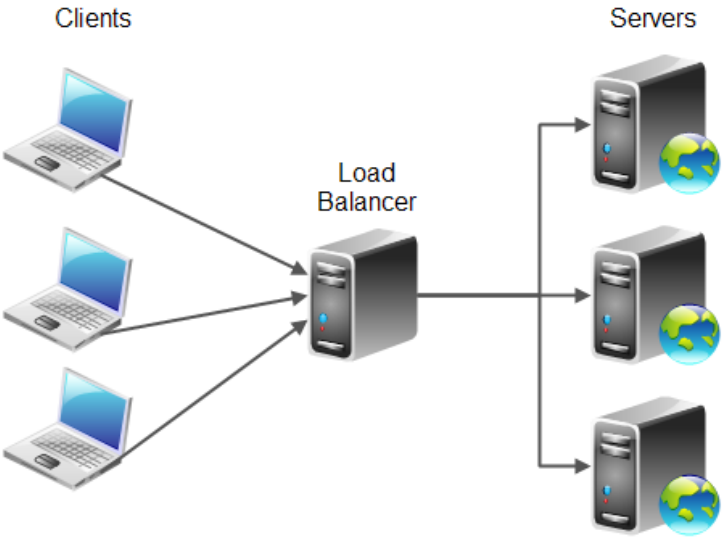

Figura 9 – Exemplo de Load Balancer

1.3.1.5 Código sob Demanda

Apesar de ser a única restrição opcional do estilo, ela permite que servidores disponibilizem código em forma de script para que seja executado no cliente. Dessa forma, extendendo a lógica de serviço do servidor para seus clientes. (**??**)

1.3.1.6 Cache

Para aumentar desempenho de um serviço. Quando um recurso é acessado por mais de um cliente, se não houve mudança neste é recomendado que estas respostas sejam armazenadas em cache, evitando o processamento desnecessário. Isso significa que servidores, quando possível, devem implementar regras de cache para beneficio de ambos os ambientes. (**??**)

1.3.2 RESTful

Muitas pessoas, ao usar diferentes URIs, verbos HTTP e formatos de representação para retorno, assumem que sua API é uma RESTful API. Tudo isso é bem importante e também considerado como parte do RESTful em si. Contudo, segundo Richardson, para ser consideirado API RESTful este precisa seguir estritamente as regras definidas pelo estilo de arquitetura REST, menos código sob demanda consideirada como opcional. Além de ter um certo level de coesão e maturidade, definido em uma escala: (**??**)

- Nível 0: É a falta de qualquer regra; diz respeito ao uso de HTTP para operações de endereços no servidor. Normalmente, usa apenas um endpoint (URI) e um verbo HTTP.
- Nível 1: Aplicação de recursos. A API é dividida em diferentes endpoints que indicam um ou mais recursos.
- Nível 2: Implementação de verbos HTTP para diferentes tipos de operações. Onde uma mesma URI pode aceitar mais de um verbo para excução de diferentes procedimentos.
- Nível 3: O conceito de HATEOS é aplicado para disponibilizar informações necessárias para interação e navegação da API.

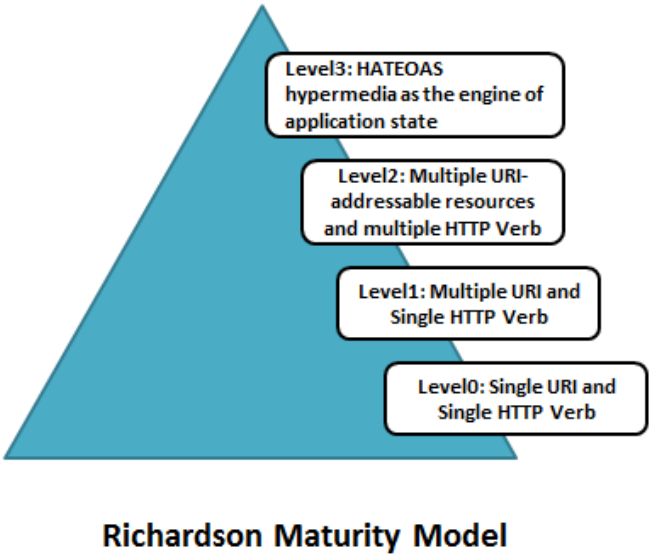

Figura 10 – Modelo de Maturidade descrito por Richardson

1.3.3 Descrições de API

Em SaaS[7], APIs REST tornaram-se padrão como interface de acesso à serviços da empresa por seus clientes. A capacidade de oferer uma completa descrição sobre a api web para permitir que usuarios descubram e entam o serviço tornou-se critico para o sucesso da empresa. Contudo, apesar do processo de implementação tournou-se uma pratica comum, a definição de metadados de apis ainda não atingiu um grau de maturidade para normas amplamente aceitas. Apenas descrever manualmente APIs através de websites em linguagem natural permite que apenas pessoas entendam, isso se for bem projetada (**??**)

Com o fracasso de formatos tradicionais para descrição de web services como WADL, a adoção duvidosa de formatos hypermedia de resposta HATEOS como HAL e a demanda cada vez mais alta por boas especificações em formato legivel por humanos e maquinas. Nos últimos

[7] Software as a Service.

anos foram introduzidas diversas ferramentas e formatos de descrição para descrever Web APIs de REST, tanto em formatos legíveis para humanos como para máquinas. (**??**)

A seguir, comparações feitas por Sandoval entre as 3 linguagens mais usadas para especificação de APIs: (**??**)

OpenAPI (Swagger)
Pros: Amplamente adotada, grande comunidade, suporte pra diversas linguagens.
Cons: Carece de especificações de metadados mais avançadas.

RAML
Pros: Suporte a especificação avançada, adoção significativa, formato legível, bom suporte da indústria.
Cons: Falta de ferramentas de auxílio, não comprovada a longo prazo.

API Blueprint
Pros: Fácil de entender, simples de escrever
Cons: Pouca adoção, Carece de especificações de metadados mais avançadas, instalação complexa.

Recentemente, empresas como Heroku tem se dedicado ao uso do JSON Schema como formato de descrição de APIs, por ser uma tecnologia limpa e ainda pouco aplicada especificamente para a construção de grandes API (**??**). Já Lynn propõe um método de modelagem de REST API em serviços IoT utilizando JSON Hyper-Schema visto nos capítulos anteriores. Com suporte a descrição de entrada e saída de dados em toda a interface, junto com descrições URIs, relações e métodos que se aplicam aos links. Além disso, o formato suporta HATEOS e serve como entrada de documentação e ferramenta para geração de código. (**??**)

1.4 GRAPHQL

GraphQL é uma linguagem de consulta de dados e interpretador de consultas para APIs. Foi desenvolvida pelo Facebook em 2012 mas sua especificação apenas publicada em 2015. Tem com objetivo fornecer uma descrição completa e compreensível de dados disponíveis em interfaces de aplicação. Além disso, permite que clientes façam consultas para busca exata de dados que desejam trabalhar. (**??**)

Importante ressaltar que GraphQL não é uma linguagem para banco de dados, por mais que possa ser utilizado para esta finalidade.

Ao invés, sua linguagem e interpretador trabalham com a idéia de mapeamento de campos e tipos de dados em aplicações, fornecendo uma interface unificada e amigável para desenvolvimento de produtos, além de uma plataforma poderosa para a construção de ferramentas. (**??**)

Por ser uma especificação, GraphQL possui implementações em diversas linguagens de programação e atualmente é usado em diversos contextos, como comunicação entre cliente-servidor, microserviços, simplificação de APIs, navegação de árvores, gerador de consultas para banco de dados, entre outros.

Para gerar um esquema GraphQL, antes é preciso definir os tipos de estruturas, seus campos de acesso e funções para mapear e retornar dados reais em código.

```
var pessoa = {
  nome: "Mateus Maso"
}

class Query {
  pessoa() {
    return pessoa
  }
}

class Pessoa {
  nome(pessoa) {
    return pessoa.nome
  }
}
```

Figura 11 – API de um código em JavaScript

```
type Query {
  pessoa: User
}

type Pessoa {
  id: ID
  nome: String
}
```

Figura 12 – Esquema GraphQL para API em JavaScript

Uma vez que o esquema GraphQL foi gerado, pode ser enviado consultas GraphQL para construção de estruturas de dados. Primeiramente, é feito a validação da consulta para garantir que esta só se refere aos tipos e campos definidos, em seguida, executa as funções fornecidas para produzir um resultado. (**??**)

```
{
  pessoa {
    nome
  }
}
```

Figura 13 – Query GraphQL para esquema

```
{
  "pessoa": {
    "nome": "Mateus Maso"
  }
}
```

Figura 14 – Resposta JSON da Query GraphQL

1.4.1 Linguagem de Consulta

Clientes que buscam realizar consultas de dados em serviços GraphQL precisam antes entender seu formato de requisição, também chamado de documento. Um documento contém operações como consultas e mutações. Além disso, é possível especificar fragmentos, uma unidade comum de composição para reuso de consultas. (**??**)

1.4.1.1 Sintaxe

A GraphQL query document describes a complete file or request string received by a GraphQL service. A document contains multiple definitions of Operations and Fragments. GraphQL query documents are only executable by a server if they contain an operation. If a document contains only one operation, that operation may be unnamed or represented in the shorthand form, which omits both the query keyword and operation name. Otherwise, if a GraphQL query document contains multiple operations, each operation must be named.

There are two types of operations that GraphQL models: query – a read-only fetch. mutation – a write followed by a fetch. Each operation is represented by an optional operation name and a selection set. If a document contains only one query operation, and that query defines no variables and contains no directives, that operation may be represented in a short-hand form which omits the query keyword and query name. An operation selects the set of information it needs, and will receive exactly that information and nothing more, avoiding over-fetching and under-fetching data. In this query, the id, firstName, and lastName fields form a selection set. Selection sets may also contain fragment references.

A selection set is primarily composed of fields. A field describes one discrete piece of information available to request within a selection set. Some fields describe complex data or relationships to other data. In order to further explore this data, a field may itself contain a selection set, allowing for deeply nested requests. All GraphQL operations must specify their selections down to fields which return scalar values to ensure an unambiguously shaped response. Fields are conceptually functions which return values, and occasionally accept arguments which alter their behavior. These arguments often map directly to function arguments within a GraphQL server's implementation.

```
{
  pessoa(id: 4) {
    id
    nome
    sobrenome
    nascimento: aniversario {
      mes
      dia
    }
    amigos(limite: 10) {
      nome
    }
  }
}
```

Figura 15 – Sintaxe

1.4.1.2 Fragmentos

Fragments are the primary unit of composition in GraphQL. Fragments allow for the reuse of common repeated selections of fields, reducing duplicated text in the document. Inline Fragments can be used directly within a selection to condition upon a type condition when querying against an interface or union.

Fragments must specify the type they apply to. In this example, friendFields can be used in the context of querying a User. Fragments cannot be specified on any input value (scalar, enumeration, or input object).Fragments can be specified on object types, interfaces, and unions.

Fragments can be defined inline within a selection set. This is done to conditionally include fields based on their runtime type. This feature of standard fragment inclusion was demonstrated in the query FragmentTyping example. We could accomplish the same thing using inline fragments.

```
{
  pessoa(id: 4) {
    ...identidade
    ... on User {
      friends {
        name
      }
    }
  }
}

fragment identidade on Pessoa {
  id
  name
}
```

Figura 16 – Fragmentos

1.4.2 Sistema de Tipagem

The GraphQL Type system describes the capabilities of a GraphQL server and is used to determine if a query is valid. The type system also describes the input types of query variables to determine if values provided at runtime are valid. A GraphQL server's capabilities are referred to as that server's "schema". A schema is defined in terms of the types and directives it supports.

A given GraphQL schema must itself be internally valid. A GraphQL schema is represented by a root type for each kind of operation: query and mutation; this determines the place in the type system where those operations begin. All types within a GraphQL schema must have unique names. No two provided types may have the same name. No provided type may have a name which conflicts with any built in types (including Scalar and Introspection types).

——-

(tipos folha: enumerados, escalares) (tipos de entrada e saida) => lista, nao nulo, objetos de entrada (tipos de composicao) => objetos (tipos de abstracao) => interfaces e unicoes

The most basic type is a Scalar. A scalar represents a primitive

value, like a string or an integer. Oftentimes, the possible responses for a scalar field are enumerable. GraphQL offers an Enum type in those cases, where the type specifies the space of valid responses. Scalars and Enums form the leaves in response trees; the intermediate levels are Object types, which define a set of fields, where each field is another type in the system, allowing the definition of arbitrary type hierarchies.

GraphQL supports two abstract types: interfaces and unions. An Interface defines a list of fields; Object types that implement that interface are guaranteed to implement those fields. Whenever the type system claims it will return an interface, it will return a valid implementing type. A Union defines a list of possible types; similar to interfaces, whenever the type system claims a union will be returned, one of the possible types will be returned.

All of the types so far are assumed to be both nullable and singular: e.g. a scalar string returns either null or a singular string. The type system might want to define that it returns a list of other types; the List type is provided for this reason, and wraps another type. Similarly, the Non-Null type wraps another type, and denotes that the result will never be null. These two types are referred to as "wrapping types"; non-wrapping types are referred to as "base types". A wrapping type has an underlying "base type", found by continually unwrapping the type until a base type is found.

```
interface Individuo {
  nome: String
}

type Pessoa implements Individuo {
  ano: Int
  foto: Foto
  amigos: [Pessoa]
}

type Foto {
  altura: Int
  largura: Int
}

union ResultadoPesquisa = Foto | Pessoa

type Pesquisa {
  resultado: ResultadoPesquisa
}
```

Figura 17 – Sistema de Tipagem

1.4.3 Instrospecção

A GraphQL server supports introspection over its schema. This schema is queried using GraphQL itself, creating a powerful platform for tool-building. Take an example query for a trivial app. In this case there is a User type with three fields: id, name, and birthday. For example, given a server with the following type definition:

```
query IntrospectionQuery {
  __schema {
    queryType { name }
    mutationType { name }
    subscriptionType { name }
    types {
      ...FullType
    }
    directives {
      name
      description
      locations
      args {
        ...InputValue
      }
    }
  }
}
fragment FullType on __Type {
  kind
  name
  description
  fields(includeDeprecated: true) {
    name
    description
    args {
      ...InputValue
    }
    type {
      ...TypeRef
    }
    isDeprecated
    deprecationReason
  }
  inputFields {
    ...InputValue
  }
  interfaces {
    ...TypeRef
  }
  enumValues(includeDeprecated: true) {
    name
    description
    isDeprecated
```

ANEXO A – Código Fonte