Mateus Maso

Um Modelo de Comunicação para Automação na Execução de Consultas de Dados sobre APIs Web

Florianópolis, SC
10 de dezembro de 2016

Mateus Maso

UM MODELO DE COMUNICAÇÃO PARA AUTOMAÇÃO NA EXECUÇÃO DE CONSULTAS DE DADOS SOBRE APIS WEB

Trabalho de Conclusão de Curso submetido ao Curso de Graduação em Sistemas de Informação para a obtenção do Grau de Bacharel em Sistemas de Informação.
Orientador: Frank Siqueira

Florianópolis, SC
10 de dezembro de 2016

À minha família.

"Time has told me not to ask for more,
someday our ocean will find its shore."

Nick Drake

RESUMO

A fim de manter a eficiência da comunicação cliente-servidor sem a necessidade do versionamento de APIs Web, serviços têm encontrado dificuldades em realizar mudanças na especificação de sua interface de acesso devido ao acoplamento causado por clientes na implementação em seu código de busca. No intuito de desenvolver uma solução para o problema encontrado, este trabalho realiza um estudo sobre o uso da linguagem GraphQL e formatos de descrição de API para propor um modelo de comunicação cliente-servidor através da automação na execução de consultas de dados. Como resultado, é desenvolvida uma ferramenta prevista pelo modelo proposto para validar sua aplicabilidade e direcionar desenvolvedores de clientes à implementação de um código de busca independente de especificação de API.

Palavras-chaves: GraphQL. REST. JSON Hyper-Schema.

ABSTRACT

In order to maintain client-server communication efficiency without the need of versioning Web APIs, services have come across problems while performing changes on their interface access specification due to the coupling caused by clients on the implementation of its fetching code. Seeking to develop a solution for the problem, this project conducts a study on the usage of GraphQL language and API description formats to propose a client-server communication model through automation in the execution of data queries. As a result, a tool foresaw by the proposed model is developed to validate its applicability and guide client-side developers to the implementation of data fetching code independent of API specification.

Key-words: GraphQL. REST. JSON Hyper-Schema.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

AST	Abstract syntax tree
API	Application Programming Interface
CPU	Central Processing Unit
JS	JavaScript
JSON	JavaScript Object Notation
HATEAOS	Hypermedia as the Engine of Application State
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
RAM	Random-access memory
REST	Representational State of Transfer
SWAPI	Starwars API
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
XML	Extensible Markup Language

SUMÁRIO

1 INTRODUÇÃO

Neste capítulo é apresentado o atual cenário do mercado de APIs Web, o problema de acoplamento encontrado na implementação do código de busca em clientes e a proposta do trabalho para solucionar este problema.

1.1 DESCRIÇÃO DO PROBLEMA

Em 2005, ocorreu uma grande transição no modelo de comunicação entre aplicações distribuídas, onde estas passaram a utilizar amplamente o protocolo HTTP e o modelo cliente-servidor para a troca de informações na World Wide Web. Um dos principais motivos que contribuiu na época para esta crescente transição foi devido a facilidade de aplicações em expor sua API através do estilo de arquitetura REST, assim como clientes em se comunicar com essas interfaces remotas. (DUVANDER, 2013)

Hoje, empresas como Facebook e Netflix mostram que, independente do estilo de arquitetura, construir APIs Web é, não apenas essencial para entrar rápido no mercado de plataformas emergentes, como também um novo meio de agregar valor em seu próprio modelo de negócio e oferecer uma melhor experiência a seus usuários. (ART, 2016)

Contudo, após anos de sua popularização, serviços têm mostrado dificuldades em realizar mudanças em suas APIs Web sem comprometer a comunicação de clientes. Isso porque clientes continuam implementando seu código de busca através de chamadas diretamente na API, ocasionando um acoplamento da especificação muitas vezes indesejado pelos serviços. Além de promover versionamento de APIs, este acoplamento tem atrasado a adoção de novas tecnologias e estilos de arquitetura por serviços.

1.2 OBJETIVOS

Com o objetivo de resolver o problema de acoplamento mencionado anteriormente, este trabalho propõe um novo modelo de comunicação cliente-servidor através da automação na execução de consultas de dados sobre APIs Web.

O modelo visa direcionar desenvolvedores de clientes à implementação de códigos de busca independente de especificação de API, resultando no desenvolvimento de clientes tolerantes às mudanças na especificação. Ainda, prevê o desenvolvimento de uma ferramenta que funciona como intermediadora na comunicação e automação de consultas, permitindo otimização de requisições e composição de serviços para consulta de dados.

Objetivos específicos

No intuito de atingir o objetivo geral do trabalho, pretende-se atender os seguintes objetivos específicos:

- Propor modelo de comunicação replicável e agnóstico à plataforma.
- Desenvolver ferramenta prevista pelo modelo para plataforma Web.
- Validar a ferramenta na comunicação entre clientes e APIs Web.
- Promover a consulta de dados através de linguagens de consulta.
- Promover a documentação de APIs através de formatos de descrição.

Limites da pesquisa

Dentro do escopo do trabalho estão delimitados os itens:

- Ênfase na comunicação entre clientes e APIs REST.
- Ênfase na tecnologia GraphQL para linguagem de consulta.
- Ênfase na tecnologia JSON Hyper-Schema para descrição de APIs.
- Testes de composição de serviços não aplicados para a ferramenta.

1.3 METODOLOGIA

A primeira etapa do trabalho foi a idealização do modelo, onde foi levantada uma série de perguntas que culminaram em um profundo estudo para comprovar se era possível resolver o problema e de que forma a solução seria implementada. Complementarmente, foram identificadas as tecnologias que poderiam ser usadas para ajudar no desenvolvimento da ferramenta proposta pelo modelo e facilitar sua replicação em diversas plataformas.

Em uma segunda etapa, foi pensado na criação de uma interface para a ferramenta que fosse simples de usar, rodasse em serviços GraphQL já existentes e pudesse ser integrado em APIs REST que já possuem algum formato de descrição de metadados.

Com o interesse de validar o protótipo do modelo, foi realizado um PoC[1]. Em paralelo, iniciou-se o processo de escrita da monografia, implementação da ferramenta e preparação de um ambiente de validação que pudesse enfatizar bem os problemas reais que clientes Web têm presenciado devido a este acoplamento na consulta de dados.

Por fim, com o objetivo de ratificar o trabalho, foram executados testes de validação, coletados os resultados e apresentados ao lado de análises e ilustrações com gráficos.

1.4 ORGANIZAÇÃO DO TEXTO

O texto está organizado em 6 capítulos. O primeiro aborda os conceitos dos fundamentos utilizados para entender o modelo, a implementação da ferramenta e o ambiente de validação. Em seguida, em um capítulo à parte, é descrita a tecnologia GraphQL, responsável por possibilitar a realização do trabalho. Após, é especificado todo o processo de desenvolvimento do trabalho através do planejamento de projeto e implementação, seguido pelo capítulo de validação onde é preparado o ambiente testes, executados e analisados através de gráficos. Por fim, no último capítulo, é exposta a conclusão sobre o trabalho, além de propor melhorias a serem desenvolvidas como trabalhos futuros.

[1] Proof of concept ou Prova de conceito

2 FUNDAMENTOS

Neste capítulo são estabelecidos os principais conceitos utilizados ao longo do trabalho. É conduzida uma breve abordagem sobre o formato JSON para representação de dados serializados na Web; o formato JSON Schema para descrição de metadados JSON; o estilo de arquitetura REST (baseado em recursos) e as atuais formas disponíveis hoje na comunidade de desenvolvimento para descrição da especificação de APIs REST.

2.1 JSON

JSON (*Javascript Object Notation*) é um formato de serialização de dados legível por humanos baseado em texto com especificação padronizada e parcialmente descritivo. Foi desenvolvido por Douglas Crockford com o objetivo de representar dados de 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 aplicações distribuídas, este formato acabou sendo amplamente utilizado em serviços como principal forma de representação de dados serializados. (DUVANDER, 2013)

O JSON, na sua essência, foi construído com base em quatro tipos primitivos de dados e outros dois para composição. Cada tipo possui seu respectivo correspondente na maioria das linguagens de programação, embora possam ser identificados por nomes diferentes. (DROETTBOOM, 2015)

Tipo	Exemplo de Valor
object	`{"chave1": "valor1", "chave2": "valor2"}`
array	`["primeiro", "segundo", "terceiro"]`
number	`1, -1, 2.9999`
string	`"Isso é uma string"`
boolean	`true, false`
null	`null`

Tabela 1 – Tipos de valores em JSON.

Através da composição de listas, objetos e tipos primitivos, consegue-se representar complexas estruturas de dados. Não existe, no entanto, um

único padrão de representação. Dada uma estrutura, é possível representá-la de inúmeras maneiras. A seguir, estão duas formas distintas de representação em JSON de uma entidade "pessoa": (DROETTBOOM, 2015)

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

Figura 1 – Representação JSON de uma entidade pessoa.

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

Figura 2 – Representação JSON alternativa de uma entidade pessoa

Ambas representações são válidas, apesar da figura 2 estar representando os dados em uma estrutura um pouco mais formal. No entanto, por ser um formato não descritivo, a responsabilidade de entender o que está sendo representado 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. (DROETTBOOM, 2015)

Para resolver isso, é abordado em seguida um dos formatos existentes hoje em dia para a descrição de estruturas JSON.

2.2 JSON SCHEMA

JSON Schema é uma linguagem de definição projetada para descrever estruturas de dados JSON por meio de esquemas. Proposta em 2009 por Kris Zyp, teve como objetivo fornecer um contrato para que aplicações soubessem como trabalhar e interagir com estruturas de dados. Por meio deste, é possível prever representações e assim realizar operações de validação, documentação, navegação por *hyperlinks* e controle de iteração. (ZYP, 2013)

Por ser uma linguagem de simples uso, para modelar um esquema basta construir um objeto JSON utilizando um subconjunto válido de chaves especiais descritas pela linguagem. No entanto, funcionalidades como descrição de estruturas multimídia[1] e a navegação de dados são apenas disponibilizadas no formato JSON Hyper-Schema, uma variação da linguagem de especificação. (JACKSON, 2016)

```
{
  "$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 3 – JSON Schema para Figura 3

[1] Capaz de reunir diversas mídias, como imagens, textos, vídeos e audio digital, por exemplo.

```
{
  "$schema": "http://json-schema.org/draft-04/hyper-schema#",
  ...,
  "properties": {
    ...,
    "foto": {
      "media": {
        "binaryEncoding": "base64",
        "type": "image/png"
      }
    }
  },
  "links": [
    {
      "rel": "foto",
      "href": "/{id}.png",
      "mediaType": "image/png"
    }
  ]
}
```

Figura 4 – JSON Hyper-Schema para Figura 3

A exemplo das figuras 3 e 4, ambos os esquemas asseguram que, dada uma estrutura JSON, para que esta seja reconhecida como uma entidade “pessoa”, deve conter as propriedades "nome"e "aniversario"com valores do tipo "string". Já na figura 4, além das estruturas definidas pela figura 3, é descrita uma nova propriedade "foto"do tipo multimídia, além de como navegar em busca desta informação.

Em casos onde a complexidade de um esquema começa a crescer, é comum a definição de sub-esquemas através da chave “definitions”. Desta forma, podem ser referenciadas pela chave "$ref"permitindo o reuso de estruturas dentro de um esquema. Vale lembrar que a chave “definitions” é apenas um mecanismo útil para definir esquemas em um lugar comum, entretanto, não sugerem que estas propriedades sejam validadas em um objeto a menos que referenciadas em outras estruturas do esquema. (LEACH, 2014)

```
{
  "$schema": "http://json-schema.org/draft-04/hyper-schema#",
  ...,
  "definitions": {
    "cidade": {
      "type": "string",
      "properties": {
        "nome": { "type": "string" },
        "estado": { "type": "string" },
        "pais": { "type": "string" }
      }
    }
  },
  "properties": {
    ...,
    "cidade": {
      "$ref": "#/definitions/cidade"
    }
  },
  "links": [
    ...,
    {
      "rel": "cidade",
      "href": "/{id}/cidade",
      "targetSchema": {
        "$ref": "#/definitions/cidade"
      }
    }
  ]
}
```

Figura 5 – JSON Hyper-Schema para Figura 4 usando $ref

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. Na tabela 2 são descritas algumas das chaves especiais usadas para descrever objetos em esquemas. (DROETT-BOOM, 2015)

De certa forma, JSON Schema continua sendo uma das únicas tentativas sérias de propor uma linguagem de definição para o formato JSON. Contudo, ainda está longe de ser considerada padrão, mas já há um número crescente de aplicações que suportam o formato, além de uma quantidade significativa de ferramentas que permitem sua validação. (PEZOA et al., 2016)

Chave	Descrição
$schema	Identificador de versão
type	Tipo de dado
title	Nome da estrutura
description	Propósito da estrutura
required	Lista de propriedades com presença obrigatória
properties	Propriedades usadas para validar uma estrutura
definitions	Propriedades (sub-esquemas) para referência
...	...
links	Lista de Link Description Objects (LDO)

Tabela 2 – Subconjunto de chaves especiais JSON Schema

Vale lembrar também que, segundo Leach, com o recente surgimento de grandes formatos de descrição de APIs ao longo dos últimos anos, JSON Hyper-Schema tem-se tornado uma ótima opção para descrever a navegação para acesso de interfaces, como por exemplo nome de método, estruturas de chamada e resposta, entre outros relacionamentos. (LEACH, 2014)

Nas próximas seções será abordado um dos estilos de arquitetura mais utilizados hoje em dia em APIs Web, além de soluções encontradas no mercado para descrição deste tipo de API em serviços.

2.3 REST

REST (*Representational State Transfer*) é um estilo de arquitetura usada para a comunicação de aplicações distribuídas 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 seis tipos de restrições que serviços deveriam aplicar para ganho de performance, escalabilidade, simplicidade, modificabilidade, visibilidade, portabilidade e confiabilidade.

Em virtude de causar grande repercussão após sua publicação, o termo REST, segundo Richardson, acabou sofrendo diversas interpretações durante o tempo e sua descrição foi representada de formas não originalmente propostas por Fielding (RICHARDSON LEONARDAMUNDSEN, 2013). Alguns descrevem que serviços que violam essas restrições não podem ser consi-

derados RESTful. Para Wildermuth, apesar de reconhecer as vantagens de cada restrição, serviços Web devem usá-los de forma pragmática. (WILDERMUTH, 2015)

Ao ser introduzido no mercado de APIs Web, REST acabou se adaptando bem por ter se mostrado uma solução de fácil acesso em clientes. Segundo Pautasso, a eliminação da complexidade existente em Web Services antes de sua publicação em 2000 fez com que REST fosse considerado um dos principais responsáveis pela popularização de arquiteturas orientada a serviços, percebido na figura 6. (PAUTASSO; ZIMMERMANN; LEYMANN, 2008)

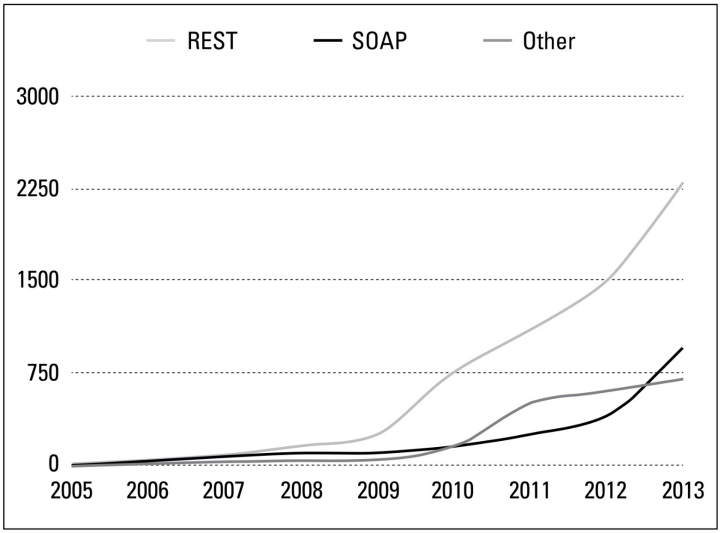

Figura 6 – Distribuição de estilos e protocolos para APIs Web

A seguir, é apresentada uma visão geral sobre as restrições propostas por Fielding para a implementação da arquitetura, além de examinar-se o impacto de cada restrição em aplicações distribuídas.

Cliente-Servidor

Nesta primeira restrição, não existe conexão entre duas aplicações dis-

tribuídas, mas sim a espera de uma (servidor) por pedidos da outra (clientes) através de chamada e resposta. O cliente (consumidor do serviço) não se preocupa com tarefas de comunicação de banco de dados, gerenciamento de cache, entre outros. O mesmo ocorre como o servidor (prestador de serviços), que não está preocupado com as tarefas do cliente como interface ou experiência do usuário, por exemplo. Isso permite a evolução independente dos dois ambientes, desde que a interface usada para comunicação entre o cliente e o servidor não seja alterada. (FIELDING, 2000)

Sem Estado

Esta restrição ajuda na viabilidade, confiabilidade e escalabilidade de aplicações distribuídas, pois garante que chamadas à API não estejam vinculadas a um determinado servidor. Como o HTTP é um protocolo sem conexão, cada requisição deve conter todas as informações necessárias para que um servidor entenda o que um cliente está executando. Para Wildermuth, no entanto, dependendo da diversidade no número de clientes, ao manter um servidor sem estado, perder-se o controle no tamanho da estrutura de resposta necessária para atender a demanda de todos os clientes. (WILDERMUTH, 2015)

Interface Uniforme

Em essência, Fielding propõe que aplicações façam o uso de verbos HTTP (POST, GET, PUT, DELETE) e identificadores uniformes de recursos (URIs) para mapear operações em APIs e minimizar o acoplamento na comunicação cliente-servidor. Essas regras de acesso são: (FIELDING, 2000)

- Identificação de Recursos: Cada recurso deve ser disponibilizado através de uma URI específica e coesa. (Exemplo: GET /customers/1)

- Manipulação de Recursos através de Representações: Um recurso pode ser representado em diferentes formatos para diferentes clientes. (Exemplo: HTML, XML, JSON)

- Resposta Auto-explicativa: Metadados devem ser adicionados na requisição e resposta de um recurso para descrever seu estado atual ou desejado. (Exemplo: código de resposta HTTP, Host, Content-Type)

- HATEOAS (*Hypermedia as the Engine of Application State*): Caso um recurso possua relacionamentos, ao ser representado, estes devem estar especificados no formato de *hyperlinks* para facilitar a navegação de dados por clientes.

Separação em Camadas

Um dos princípios desta restrição está em evitar que clientes façam diretamente requisição para o servidor sem antes passar por um intermediário, como por exemplo um balanceador de carga (*load balancer*)[2]. Desse modo, fica assegurado que clientes apenas se preocupem com a comunicação, deixando que intermediários lidem com a distribuição de requisições. (FIELDING, 2000)

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. Com isso, a lógica de serviço do servidor é estendida para seus clientes. (FIELDING, 2000)

Cache

Visa aumentar o desempenho de um serviço. Quando um recurso é acessado por mais de um cliente, se não houver mudança é recomendado que a resposta seja armazenada em cache, evitando o processamento desnecessário. Isso significa que servidores, quando possível, devem implementar regras de cache para beneficio de ambos os ambientes. (FIELDING, 2000)

[2] Técnica para distribuir a carga de trabalho uniformemente entre dois ou mais computadores

2.4 DESCRIÇÃO DE API REST

Atualmente, o processo de descrição de APIs tem se tornado um dos principais fatores de sucesso na aceitação de serviços por desenvolvedores. No entanto, diferentemente do processo de implementação, a prática de descrição de APIs REST ainda continua sendo conduzida em sua maioria manualmente através de linguagem natural. Isso porque o REST não apresenta uma forma de documentação externa para descrição de metadados. (LUCKY et al., 2016)

Em oposição a isso, Fielding propõe a descrição dinâmica de APIs através do uso de *hyperlinks* na representação de recursos para navegação de dados (HATEOAS). Contudo, para Knupp, a solução proposta por Fielding é questionável, pois na sua visão dificulta a legibilidade da interface de acesso, não prevê documentação, cria complexidade de implementação e aumenta de forma significativa o tamanho de resposta. (KNUPP, 2016)

Em busca de oferecer uma solução simples e completa para descrição de APIs REST, foram introduzidas nos últimos anos diversas soluções por empresas e comunidades de desenvolvimento. A seguir, são descritas três das linguagens e formatos que ganharam maior popularidade devido a sua facilidade de uso e legibilidade por humanos e máquinas. (SANDOVAL, 2015)

OpenAPI
- **+** Amplamente adotada, ampla comunidade, suporte a diversas linguagens.
- – Carece de especificações avançadas de metadados.

RAML
- **+** Suporte à especificação avançada de metadados, adoção significativa, formato legível, suporte da indústria.
- – Falta de ferramentas de auxílio, não comprovada a longo prazo.

API Blueprint
- **+** Fácil de entender e simples de escrever.
- – Pouca adoção, carece de especificações avançadas de metadados, difícil de executar.

Como ainda não existe uma padronização de formato (externo) para descrição de APIs, novas tecnologias estão sujeitas a emergir para melhor se adaptar a comunidade de desenvolvimento. Uma delas, não mencionada na lista, é o JSON Hyper-Schema. Através de Lynn e Leach, recentemente mostrou ser um método simples e completo para modelagem de APIs REST, uma vez que possui suporte à descrição de representação de entrada e saída, relacionamentos, HATEOAS, URIs e verbos HTTP. (LYNN; DORNIN, 2016; LEACH, 2014)

3 GRAPHQL

O GraphQL é uma linguagem de consulta de dados, além de um interpretador, criada pelo Facebook para trabalhar com APIs de forma alternativa. Seu objetivo é fornecer uma descrição completa e compreensível de dados disponíveis em interfaces de aplicação, permitindo que clientes façam consultas de dados que desejam trabalhar de forma precisa. (FACEBOOK, 2016)

Por ser uma especificação recente, após sua publicação em 2015, o GraphQL já apresenta implementações em diversas linguagens de programação e atualmente é utilizado em diversos contextos, como na comunicação cliente-servidor, microsserviços, navegação de árvores, gerador de consultas para banco de dados, entre outros.

É importante ressaltar que o GraphQL não é uma linguagem de banco de dados, por mais que possa ser utilizada para esta finalidade. Ao invés, sua linguagem e interpretador trabalham com a ideia de mapeamento de campos e tipos de dados de retorno em APIs, fornecendo um esquema para interagir com interfaces através da execução de consultas da linguagem. (FACEBOOK, 2016)

Nas figuras 7 e 8, é descrito um exemplo de mapeamento de uma API em JavaScript em um esquema GraphQL, através da análise de estruturas de retorno.

```
var pessoa = {
  id: 1,
  nome: "Mateus Maso"
}

class Query {
  pessoa() {
    return pessoa
  }
}

class Pessoa {
  id(pessoa) {
    return pessoa.id
  }

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

Figura 7 – API JavaScript escrito em ES6

```
type Query {
  pessoa: Pessoa
}

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

Figura 8 – Esquema GraphQL para API da Figura 8

Após a criação do esquema, as figuras 9 e 10 descrevem um exemplo de consulta utilizando a sintaxe do GraphQL e os tipos de dados mapeados pelo esquema. Ao ser executada, o interpretador (esquema) irá analisar, validar e transformar a consulta em chamadas para a API, retornado uma representação exata dos dados requisitados no formato JSON. (FACEBOOK, 2016)

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

Figura 9 – Consulta GraphQL para Figura 9

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

Figura 10 – Resposta JSON esperada após execução da Figura 10

A seguir, é abordado os elementos que compõem a linguagem, seu sistema de tipagem e o processo de introspecção.

3.1 LINGUAGEM DE CONSULTA

Clientes que buscam realizar consultas de dados em esquemas GraphQL, antes precisam entender seu documento de requisição. Um documento GraphQL contém operações de consultas ou mutações, além de unidades de composição e reuso de consultas, descritas pelo nome de fragmentos. (FACEBOOK, 2016)

Sintaxe

Documentos GraphQL são inspirados no formato de estrutura de dados JSON, porém sem seus valores e com alterações na sintaxe. Para que um esquema entenda um documento GraphQL, este deve descrever ao menos uma operação de consulta ou mutação. Uma operação de consulta é um pro-

cesso de leitura da API. Já uma operação de mutação é representada por dois processos, uma escrita seguida de uma leitura na API.

Ao executar uma operação, deve-se expressar um conjunto de dados (seleção) que se deseja receber. Este conjunto é representado por campos e fragmentos, visto na figura 11, onde um campo pode receber argumentos e deve descrever um dado ou subconjunto de dados. Isso permite explorar relacionamentos complexos através do profundo aninhamento de conjuntos de seleções em busca de retornar uma estrutura JSON parecida com o que se escreve na linguagem.

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

Figura 11 – Seleção de campos em consultas GraphQL

Fragmentos

Fragmentos são a principal unidade de composição em GraphQL, pois permitem o reuso de seleção de campos que se repetem em documentos. Um fragmento é representado por um nome, seguido pelo tipo que está sendo aplicado à seleção de campos. Podem também ser expressos de forma "inline", onde não há a necessidade de definir um nome.

```
query {
  pessoa(id: 4) {
    id
    ...identidade
    ...relacionamentos
  }
}

fragment identidade on Pessoa {
  nome
  sobrenome
  nascimento: aniversario {
    mes
    dia
  }
}

fragment relacionamentos on Pessoa {
  amigos(limite: 10) {
    nome
  }
}
```

Figura 12 – Consulta da Figura 12 utilizando fragmentos

3.2 SISTEMA DE TIPAGEM

Para descrever um esquema GraphQL a partir de uma API, antes é preciso fazer o uso de seu sistema de tipagem. Através da abstração de entidades de uma API, representa-se um conjunto finito de tipos para serem acessados por documentos GraphQL. Um tipo é uma forma de representação de dados específico da linguagem de definição, que indica à ferramenta como interpretar operações de consulta e mutação.

Durante o processo de mapeamento de entidades, faz-se a conversão de estruturas de dados em conjuntos dos tipos folha, de acondicionamento, de composição e abstratos. Um tipo folha é representado por valores finais, singulares e não nulos da estrutura de retorno, onde os tipos de acondicionamento e composição são utilizados em cima dos tipos folha para alterar o comportamento e combinar outros tipos em busca da representação de estruturas mais complexas. Por fim, existem os tipos abstratos, que servem para

reusar os tipos anteriores através do uso de conceitos de alto nível.

Tipo	Classificação
Enum	Folha
Int	Folha
Float	Folha
String	Folha
Boolean	Folha
ID	Folha
Object	Composição
Union	Abstrato
Interface	Abstrato
Non-Null	Acondicionamento
List	Acondicionamento

Tabela 3 – Classificação de tipos GraphQL

Após a conversão de estruturas de dados em tipos GraphQL, para que estes também sejam acessados por documentos, é preciso incluí-los em um tipo de composição especial definido pelo esquema chamado "root". Dessa forma, é possível modelar os primeiros campos de acesso que o esquema deseja disponibilizar para que clientes façam a execução de operações a partir deles.

```
interface Individuo {
  nome: String
}

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

type Foto {
  url: String
}

union Resultado = Foto | Pessoa

type Pesquisa {
  resultado: Resultado
}
```

Figura 13 – Esquema GraphQL utilizando listas, interfaces e uniões

3.3 INTROSPECÇÃO

Em GraphQL, o termo introspecção refere-se a uma consulta especial usada na busca por metadados em esquemas. Uma das vantagens desta funcionalidade em aplicações distribuídas está no seu uso para descrever serviços que expõe esquemas GraphQL para acesso de clientes. Estes, por sua vez, podem usar a introspecção para inspecionar documentação, alertar sobre campos obsoletos (*deprecated*), gerar código, entre outras possibilidades.

```
query introspeccao {
  __schema {
    queryType { name }
    mutationType { name }
    types {
      kind
      name
      description
      fields {
        name
        description
      }
    }
  }
}
```

Figura 14 – Consulta de introspecção

4 DESENVOLVIMENTO

Neste capítulo é descrito o processo de planejamento de projeto, o modelo de comunicação proposto e a implementação da ferramenta prevista pelo modelo.

4.1 PLANEJAMENTO DE PROJETO

Para que clientes possam trabalhar com dados remotos, é preciso que haja um meio de buscá-los antes de executar qualquer lógica que dependa deles. Para isso, a solução comumente adotada é a implementação de um código de busca para acesso remoto de dados em APIs de serviços[1].

Para que se possa manter garantia na comunicação, ao passar do tempo, fez-se necessário estabelecer um contrato de acesso entre o cliente e a interface do serviço. Isso porque a atual implementação do código de busca por clientes não prevê mudanças na especificação da API. Este contrato é, portanto, representado no modelo de comunicação da figura 15 através de chamadas entre o código de busca do cliente e a API do serviço.

[1] Infraestrutura distribuída (servidores, banco de dados, etc) que respondem a pedidos de operações oriundas de clientes em forma de requisições de API.

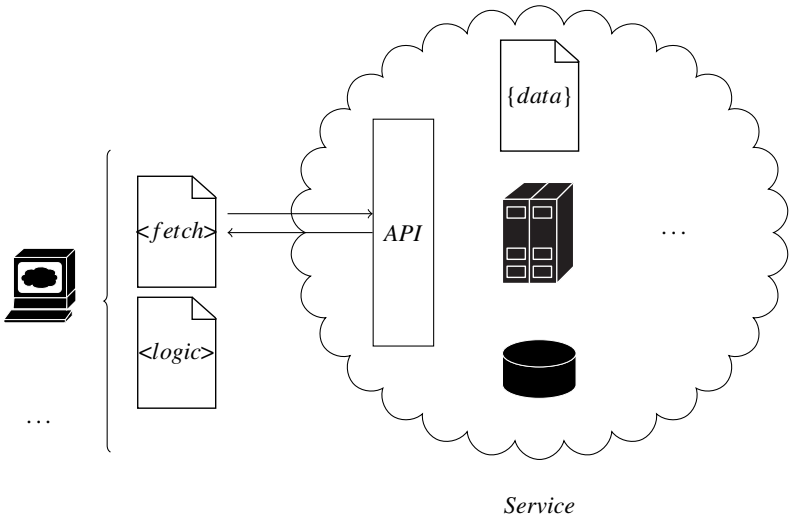

Figura 15 – Modelo de comunicação entre cliente e serviço

O modelo de comunicação representado pode não revelar problemas para serviços com pouca demanda de acesso e diversidade de consultas. Contudo, ao longo do tempo e com o aumento na quantidade de contratos, alterações na especificação como, por exemplo, as de fluxo de dados[2] podem se tornar um desafio.

Mudanças no fluxo de dados pela API são inevitáveis em aplicações distribuídas. Elas ocorrem para abraçar a constante transformação por clientes na execução de consultas de dados. Com isso, busca-se manter uma comunicação eficiente através da redução no número de chamadas executadas na API e do tamanho de dados transmitidos pela rede. Essas mudanças podem ser desde uma simples alteração no nome de um método ou número argumentos, até as mais complexas, como dados de retorno e estilo de arquitetura. A figura 16 ilustra o fluxo de dados entre um cliente e um serviço.

[2] Fluxo de acesso por clientes para consulta de dados sobre APIs de serviços.

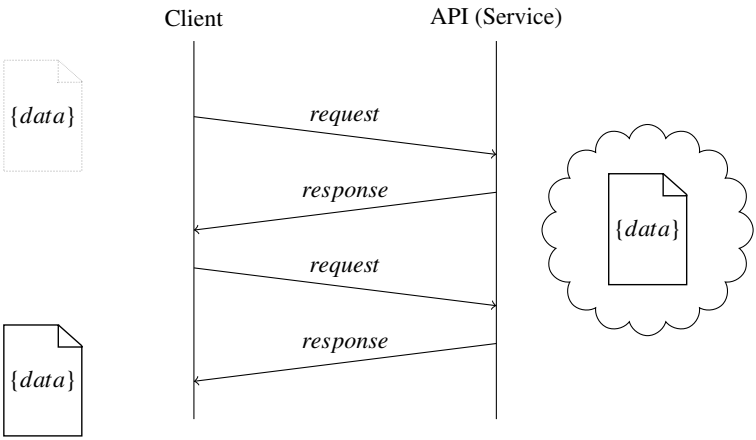

Figura 16 – Fluxo de dados entre cliente e API

O impacto causado em clientes por mudanças no fluxo de dados de uma API felizmente é previsível, pois afeta diretamente, em sua maioria, o código de busca. Por outro lado, mudanças como a alteração de campos nas estruturas de dados, como renomear um campo "nome"para "nome_completo", são de um grau de complexidade maior, pois pode não ter impacto no código de busca e sim na lógica da aplicação, onde é bem mais difícil de depurar erros.

Com o objetivo de viabilizar um novo modelo de comunicação a fim de evitar o impacto no código de busca em clientes devido a mudanças no fluxo de dados, fez-se necessário solucionar os seguintes problemas descritos na tabela 4.

Problema	Solução
(1) Garantir que não haja criação de contrato entre código de busca e API.	(1) Construir um intermediador responsável por manter a comunicação entre cliente e serviço.
(2) Realizar mudanças no fluxo de dados de uma API sem interferir no código de busca dos clientes.	(2) Evitar que clientes escrevam código de busca voltado à especificação da API, e sim para o intermediador.
(3) Encontrar uma forma de expressar requisições entre o cliente e o intermediador.	(3) Usar uma linguagem de consulta para expressar dependência de dados.
(4) Mapear consultas no intermediador em requisições para API de serviços.	(4) Analisar dependência de dados e formas de acesso através de metadados da API de serviços.

Tabela 4 – Problema-solução na concepção do novo modelo

4.2 PROPOSTA DE MODELO

Um novo modelo é proposto com o intuito de melhorar a comunicação cliente-servidor apresentada. Observado na figura 17, este prevê a criação de uma ferramenta no cliente para a intermediação da comunicação entre o código de busca e a API de serviços. Além disso, há a necessidade de reimplementação do código de busca para uma nova linguagem de consulta que seja interpretada pelo intermediador. Da mesma forma, soma-se a criação de um arquivo no serviço para descrição de metadados da API e outro no cliente para configuração, ambos essenciais para o funcionamento da ferramenta.

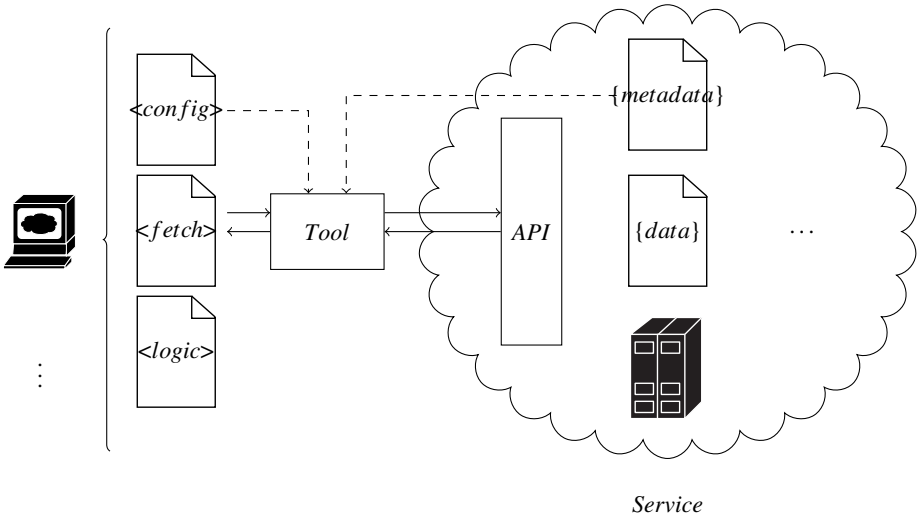

Figura 17 – Modelo proposto para evitar contrato de comunicação

A proposta de eliminação de contrato visa também automatizar o acesso de clientes em APIs. Através da implementação de algoritmos na ferramenta, escolhe-se o caminho de acesso de melhor custo-benefício em relação aos serviços disponíveis para o cliente. O modelo proporciona, além disso, um ambiente escalável para consulta de dados através da composição de serviços, visto na figura 18.

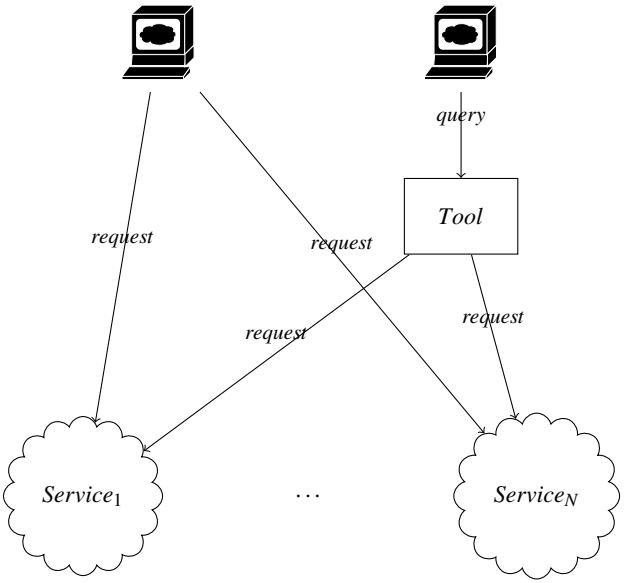

Figura 18 – Composição de serviços através da ferramenta

4.3 ESPECIFICAÇÃO DA FERRAMENTA

Visando a aceitação da ferramenta prevista pelo modelo em ambientes de desenvolvimento de clientes, foi preciso pensar em uma especificação que apresentasse uma interface de baixa curva de aprendizagem, um fluxo de execução replicável e agnóstico à plataforma, além do baixo impacto na base de código de clientes. Por conseguinte, a ferramenta proposta foi desenvolvida pensando na reutilização de tecnologias promissoras ou bem consolidadas no mercado de desenvolvimento.

Com a sucessão de estudos, a escolha foi em utilizar a tecnologia GraphQL como principal biblioteca para ajudar com intermediação da comunicação do modelo proposto. Ao invés da tecnologia Falcor (apontada durante o estudo), GraphQL foi o único que apresentou uma solução robusta que permitisse o acesso de APIs Web por clientes através de consultas em seu código de busca. Da mesma forma, buscou-se trabalhar com formatos abertos de descrição de metadados de APIs, como por exemplo o JSON Hyper-

Schema. Através de adaptadores, a ferramenta permite acelerar a integração de serviços que já oferecem metadados em formatos suportados.

A seguir, é realizada a análise dos dois fluxos de execução descritos na especificação da ferramenta. O primeiro, ilustrado na figura 19, é o processo de criação do intermediador, que recebe de entrada funções de configuração de serviços e retorna um esquema GraphQL. O segundo, mostrado na figura 20, é o processo de consulta de dados no intermediador, que recebe de entrada consultas escritas na sintaxe da linguagem GraphQL e, após chamadas às APIs, retorna estruturas de dados JSON.

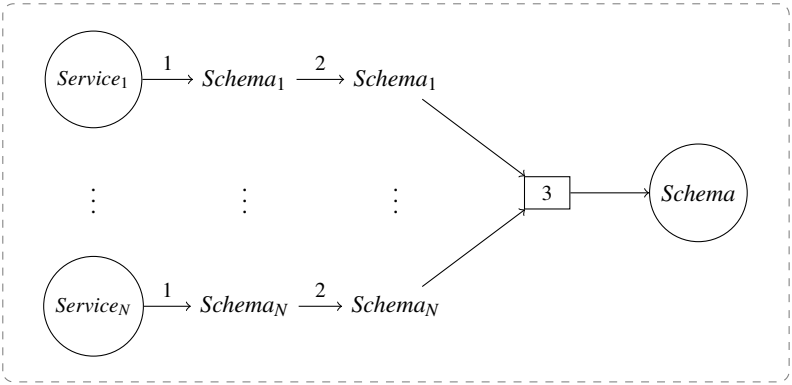

Figura 19 – Processo de criação do intermediador

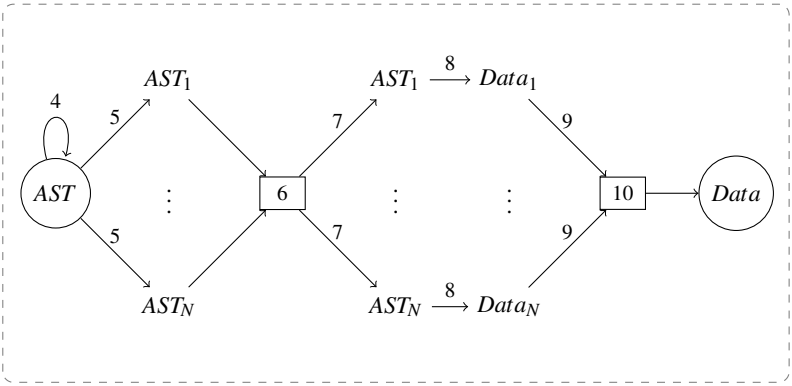

Figura 20 – Processo de consulta de dados no intermediador

Os passos enumerados nas figuras 19 e 20 são descritos a seguir:

1. Constrói um esquema GraphQL para o serviço através de seus metadados. Este processo pode ser sobrescrito por adaptadores.

2. Envolve os campos do esquema GraphQL gerado para o serviço visando facilitar a consulta e, depois, unificar com outros esquemas de maneira correta.

3. Realiza a união dos esquemas GraphQL gerados para cada serviço. Cria um esquema unificado para que a ferramenta possa interpretar consultas de diversos serviços em uma única operação.

4. Simplifica o AST gerado pela consulta no esquema unificado para facilitar o trabalho na resolução de fragmentos GraphQL.

5. Transforma o AST principal da consulta em um AST especifico para cada esquema de serviço. Dessa forma, separa os dados que cada serviço consegue disponibilizar para o cliente. Este processo pode ser sobrescrito por adaptadores.

6. Analisa os ASTs gerados para cada serviço na etapa de transformação e, a partir de heurísticas, reduz os campos dos ASTs para otimizar consultas. A heurística envolve o número de requisições necessárias para número de dados retornados e requisições para obtê-los.

7. Desfaz o envoltório do AST para poder realizar a requisição correta na consulta dos dados do serviço.

8. Busca os dados a partir do AST, onde interpreta os metadados do serviço e realiza requisições para a API do serviço. Este processo pode ser sobrescrito por adaptadores.

9. Recria o envoltório dos dados retornados para que o esquema GraphQL entenda e consiga unificar com outros dados de maneira correta.

10. Unifica os dados retornados pelas APIs de cada serviço, nas quais foram feitas as requisições.

Em resumo, o processo de criação do intermediador começa através da resolução das funções de configuração de serviços (URL, metadados, adaptador e definições de envoltório) para construir um esquema GraphQL de cada um. Na sequência, é aplicado o envoltório dos campos dos esquemas, estendendo-se até gerar um esquema GraphQL unificado.

Com relação ao processo de consulta de dados no intermediador, este começa através de uma chamada de consulta no esquema gerado. Primeiramente utiliza-se o AST fornecido pela função de resolução de consulta GraphQL e converte-se para uma estrutura de maior facilidade de trabalho. Em seguida, transforma-se o AST já simplificado em ASTs específicos para o esquema de cada serviço. Durante esse processo, reduz-se as árvores através de um algoritmo de otimização de busca que analisa o conjunto como um todo. Por fim, para se poder realizar a consulta de dados correspondente em cada serviço, é desfeito o envoltório dos ASTs, são realizadas as requisições, é criado o envoltório e é feita a união dos dados JSON retornados.

4.4 IMPLEMENTAÇÃO DA FERRAMENTA

A fim de aplicar a especificação da ferramenta abordada no planejamento de projeto, foi proposta a implementação de onze funções, divididas em três categorias, para o uso de clientes na plataforma Web. Cada uma dessas funções levantadas resolve um problema com o intuito de chegar ao objetivo de representar os dois fluxos de execução.

Dentro do conjunto de funções, quatro delas são responsáveis por criar o intermediador; três funções analisam as consultas através do formato AST; e as outras quatro transformam as consultas em requisições para API de serviços. Somente quatro são públicas e expostas para o cliente, sendo uma delas (composeSchema) a principal para criação do esquema de consulta, e as outras três (buildSchema, transformAST, fetchData) para serem sobrescritas por adaptadores.

A seguir, são descritas brevemente essas três categorias de funções, ao lado de seu objetivo e a assinatura JavaScript de cada função em Flow[3].

[3] Verificador de tipo estático para JavaScript.

Criação do esquema

Consiste de quatro funções e seu principal objetivo é compor, de forma assíncrona, um esquema GraphQL unificado a partir das funcões de configuração de cada serviço.

```
function composeSchema(
  services: [Service]
): Promise<GraphQLSchema>

function buildSchema(
  metadata: JSON
): Promise<GraphQLSchema>

function wrapSchema(
  schema: GraphQLSchema,
  wrapper: JSON
): Promise<GraphQLSchema>

function deepExtendSchema(
  schemas: [GraphQLSchema]
): Promise<GraphQLSchema>
```

Figura 21 – Assinatura JS das funções para criação do esquema

Análise de consulta

É constituído de três funções que buscam analisar consultas GraphQL executadas no esquema gerado em formato AST. A análise descreve algoritmos que simplificam, transformam e reduzem o AST principal da consulta, quebrando-o em ASTs específicos para cada serviço.

```
function simplifyAST(
  value: AST,
  info: JSON
): SimplifiedAST

function transformAST(
  metadata: JSON,
  schema: GraphQLSchema,
  ast: SimplifiedAST
): SimplifiedAST

function reduceASTs(
  rootAST: SimplifiedAST,
  asts: [SimplifiedAST]
): [SimplifiedAST]
```

Figura 22 – Assinatura JS das funções para análise de consulta

Consulta de dados

Consiste as quatro últimas funções responsáveis por converter os ASTs de cada serviço e realizar as respectivas requisições para consulta de dados sobre APIs.

```
function unwrapAST(
  ast: SimplifiedAST,
  schema: GraphQLSchema,
  wrapper: JSON
): SimplifiedAST

function fetchData(
  metadata: JSON,
  ast: SimplifiedAST,
  url: String
): Promise<JSON>

function wrapData(
  data: JSON,
  schema: GraphQLSchema,
  wrapper: JSON
): JSON

function deepExtendData(
  data: [JSON]
): JSON
```

Figura 23 – Assinatura JS das funções para consulta de dados

5 VALIDAÇÃO

Com o objetivo de validar o modelo proposto, é realizada uma pesquisa comparativa com foco no impacto da relação do uso da ferramenta implementada e mudanças no fluxo de dados sobre APIs Web. Para isso, é desenvolvido um ambiente de validação onde dois clientes, um com a ferramenta e o outro não, realizam três consultas de dados sobre uma API REST. Por fim, é aplicada uma série de quatro mudanças no fluxo de dados da API e coletados os dados para análise do impacto causado no código de busca desses clientes.

Escopo de serviço

Escolheu-se trabalhar com um escopo de serviço conhecido como SWAPI (*Starwars API*), bastante utilizado para análise de implementações de tecnologias e estilos de arquitetura em linguagens de programação. Nele, são descritas seis entidades (Pessoa, Filme, Espaçonave, Veículo, Espécie, Planeta) e seus respectivos relacionamentos. Dentro do escopo de serviço, é implementada uma API REST que expõe consultas dessas entidades em formato de representação JSON através de URIs, listadas na tabela 5. Nota-se que as representações possuem apenas um nível de expansão, ou seja, para acesso aos relacionamentos é preciso consultar a API através de links descritos pela estrutura de retorno.

URI	Descrição
/people	Busca lista de pessoas
/people/:id	Busca pessoa pelo id
/films	Busca lista de filmes
/films/:id	Busca filme pelo id
/starships	Busca lista de espaçonaves
/starships/:id	Busca espaçonave pelo id
/vehicles	Busca lista de veículos
/vehicles/:id	Busca veículo pelo id
/species	Busca lista de espécies
/species/:id	Busca espécie pelo id
/planets	Busca lista de planetas
/planets/:id	Busca planeta pelo id

Tabela 5 – Descrição das URIs SWAPI

Perguntas e respostas esperadas

Com o propósito de explorar cada URI para consulta de dados na SWAPI, foram determinadas três perguntas onde fossem envolvidas pelo menos três das entidades do escopo. Para cada pergunta existe apenas uma resposta certa e sua lógica é baseada em campos das estruturas de dados de retorno.

Q1. Qual o nome do filme no qual aparecem mais personagens oriundos de um planeta deserto? **R:** "Revenge of the Sith"

Q2. Qual o nome da espécie predominante entre os habitantes do planeta "Tatooine"? **R:** "Droid"

Q3. Qual o nome do personagem que mais pilota espaçonaves e veículos durante o filme "A New Hope"? **R:** "Chewbacca"

No intuito de atingir as respostas esperadas, a tabela 6 descreve o fluxo de dados necessário para responder cada pergunta através da consulta de dados na SWAPI.

Pergunta	Requisições	Número de chamadas
Q1	**GET** /api/films	**x1** films
	GET /api/people/:id	**x162** films.characters
	GET /api/planet/:id	**x162** films.characters.homeworld
Q2	**GET** /api/planets/1	**x1** planet
	GET /api/people/:id	**x10** planet.residents
	GET /api/species/:id	**x2** planet.residents.species
Q3	**GET** /api/films/1	**x1** film
	GET /api/starships/:id	**x8** film.starships
	GET /api/people/:id	**x9** film.starships.pilots
	GET /api/vehicles/:id	**x4** film.vehicles
	GET /api/people/:id	**x0** film.vehicles.pilots

Tabela 6 – Fluxo de dados para responder as perguntas

Consultas GraphQL e metadados

Para permitir a comunicação entre o cliente que faz o uso da ferramenta com a SWAPI, foi preciso implementar três consultas GraphQL (uma para cada pergunta), além de descrever os metadados da API REST em formato JSON Hyper-Schema. Na figura 24, foram selecionados apenas os campos necessários para o funcionamento da lógica de resposta. Em relação aos metadados da figura 25, foram descritas apenas as URIs utilizadas.

```
query q1 {
  allFilms {
    title
    characters {
      homeworld {
        climate
      }
    }
  }
}

query q2 {
  planet(planetID: 1) {
    residents {
      species {
        name
      }
    }
  }
}

query q3 {
  film(filmID: 1) {
    starships {
      pilots {
        name
      }
    }
    vehicles {
      pilots {
        name
      }
    }
  }
}
```

Figura 24 – Consultas GraphQL para as perguntas

```
{
  "$schema": "http://json-schema.org/draft-04/hyper-schema#",
  "title": "swapi",
  "type": "object",
  "definitions": {
    "allFilms": { ... },
    "film": { ... },
    "people": { ... },
    "planet": { ... },
    "species": { ... },
    "starship": { ... },
    "vehicle": { ... }
  },
  "properties": {
    "allFilms": { "$ref": "#/definitions/allFilms" },
    "film": { "$ref": "#/definitions/film" },
    "people": { "$ref": "#/definitions/people" },
    "planet": { "$ref": "#/definitions/planet" },
    "species": { "$ref": "#/definitions/species" },
    "starship": { "$ref": "#/definitions/starship" }
  },
  "links": [{
    "rel": "allFilms",
    "href": "/films",
    "targetSchema": {
      "$ref": "#/definitions/allFilms"
    }
  }, {
    "rel": "film",
    "href": "/films/{filmID}",
    "schema": { ... },
    "targetSchema": {
      "$ref": "#/definitions/film"
    }
  }, {
    "rel": "planet",
    "href": "/planets/{planetID}",
    "schema": { ... },
    "targetSchema": {
      "$ref": "#/definitions/planet"
    }
  }]
}
```

Figura 25 – JSON Hyper-Schema para SWAPI

Mudanças no fluxo de dados

Para avaliar o impacto no código em ambos clientes, são propostos quatro tipos de mudanças não acumulativas na especificação da API que afetam o fluxo de dados. Cada uma busca por em teste a capacidade da ferramenta em se adaptar e realizar a comunicação mesmo após a alteração no fluxo. Nota-se que, para cada mudança na API do serviço, é preciso a atualização de seus metadados para a correta operação da ferramenta. A tabela 7 descreve o *changelog* das mudanças realizadas.

Mudança	Descrição	Changelog
C1	Mudança de acesso em endereços.	• Renomeação da URI /films para /movies. • Renomeação da URI /films/:id para /movies/:id
C2	Mudança no nível da estrutura de resposta em endereços.	• Expansão do campo pilots da URI /starships/:id • Expansão do campo pilots da URI /vehicles/:id
C3	Introdução de novos endereços.	+ Adição da URI /tatooine. + Adição da URI /films/:id/characters.
C4	Substituição de endereço *deprecated*.	+ Adição da URI /people/:id/homeworld. – Remoção da URI /planet/:id.

Tabela 7 – Changelog do novo fluxo de dados

Variáveis

São descritas cinco variáveis para análise dos quatro testes de validação em cada um dos clientes. Todas são quantitativas e coletadas no cliente após cada processo de mudança da API. Dessa maneira, totalizam um número de 40 dados normalizados possíveis para análise no final da execução nos dois clientes.

Variável	Unidade	Tipo
Porcentagem de acerto	%	Quantitativa
Tamanho de resposta	kb	Quantitativa
Número de requisições	inteiro	Quantitativa
Tempo de busca de metadados	ms	Quantitativa
Tempo de processamento	ms	Quantitativa

Tabela 8 – Variáveis de coleta e análise

5.1 RESULTADOS

Os resultados obtidos nos testes de validação apresentam a média dos valores coletados após diversas execuções dos quatro testes de mudança de forma sequencial em uma máquina virtual. Tanto o serviço SWAPI como os clientes JavaScript foram executados no mesmo ambiente computacional, porém com conexão local de latência 10Mbps para simulação de um ambiente distribuído. Para a variável de processamento, foram utilizados valores que pudessem representar um dispositivo computacional de médio porte contando com 1 CPU de 2.4 GHz e 4Gb de RAM.

Inicialmente, observa-se na figura 26 que o cliente 1 (sem o uso da ferramenta) não apresenta um bom índice de acerto das respostas. Dentre as quatro mudanças, obteve apenas 58% de acerto, sendo a C3 a única mudança em que conseguiu responder corretamente todas as perguntas, pois não houve mudança nas URIs que estava utilizando. Um resultado esperado, significando que houve quebra de contrato e impacto negativo no código de busca.

Em contrapartida, o cliente 2 (com o uso da ferramenta) apresenta um resultado no índice de acerto da figura 27 36.61% superior ao cliente 1. Com um total de 91,5% de acerto, apenas não completou com 100% de acerto pois a mudança C4 não permitiu que fosse possível acessar todos os dados necessários para a responder da pergunta Q2. Isso representa que a ferramenta foi capaz, através do intermediador, de evitar a criação de contrato e causar impacto negativo ao código de busca.

Nota-se que ambos os clientes não apresentaram erros na resposta, porém acabam não respondendo pois ocorrem exceções durante o código de busca ou na lógica do cliente devido ao impacto das mudanças no fluxo de

dados.

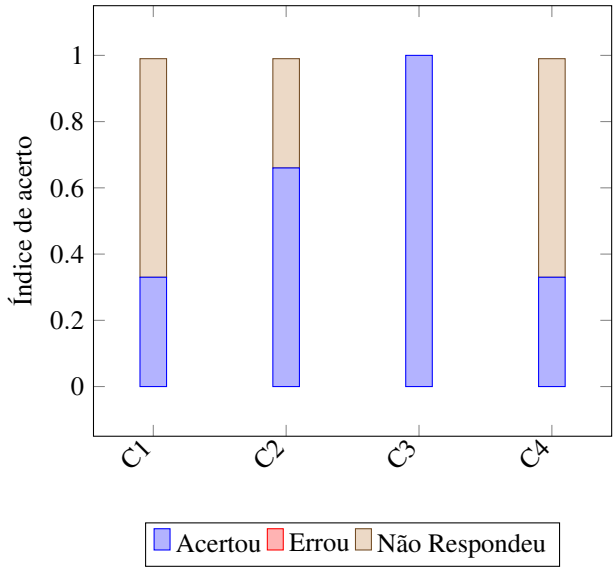

Figura 26 – Índice de acerto sem o uso da ferramenta

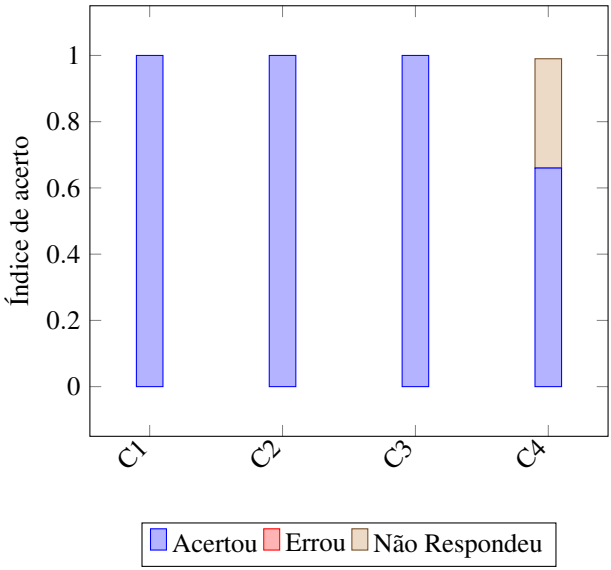

Figura 27 – Índice de acerto com o uso da ferramenta

Ao analisar a mudança C3 isoladamente, onde ambos conseguem responder com 100% de acerto, percebe-se nas figuras 28 e 29 que o cliente 2 consegue realizar uma consulta com melhor desempenho (menor número de requisições e tamanho de dados) que o cliente 1. Isso porque, após a mudança e sem alterar o código de busca de dados, a ferramenta consegue remapear as requisições geradas pelas consultas GraphQL graças ao algoritmo que automatiza as chamadas à API.

Outro ponto importante é que, após a mudança C3 e a atualização dos metadados no cliente, o algoritmo da ferramenta percebe que há a possibilidade de realizar menos requisições em busca dos dados para responder as perguntas. Isso resulta em uma redução de aproximadamente 54% dos acessos entre o cliente 1 e cliente 2.

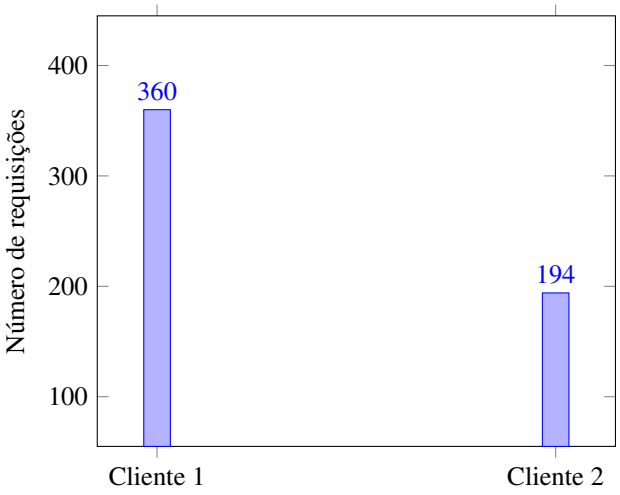

Figura 28 – Comparação no número de requisições C3

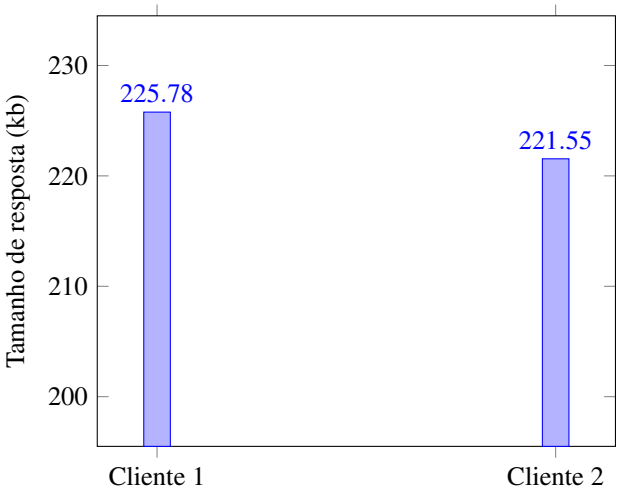

Figura 29 – Comparação no tamanho de resposta C3

Apesar dos ganhos de performance e desenvolvimento através do uso da ferramenta vistos anteriormente, percebe-se na figura 30 um outro cenário

que indica o lado negativo do seu uso causado pelo tempo de *overhead*[1]. Em média, a ferramenta atrasou em 1.7 segundos a execução na consulta de dados do cliente, sendo o tempo de busca de metadados responsável por somar mais da metade deste atraso inicial.

Contudo, é importante levar em consideração que este tempo de *overhead* é um impasse relativo ao tempo de vida do cliente que está sendo executado. Por exemplo, em clientes com tempo de vida curto ou que dependem de carregamento rápido, a ferramenta pode não ser a solução ideal. Nos demais casos, são visíveis os benefícios que seu uso pode trazer.

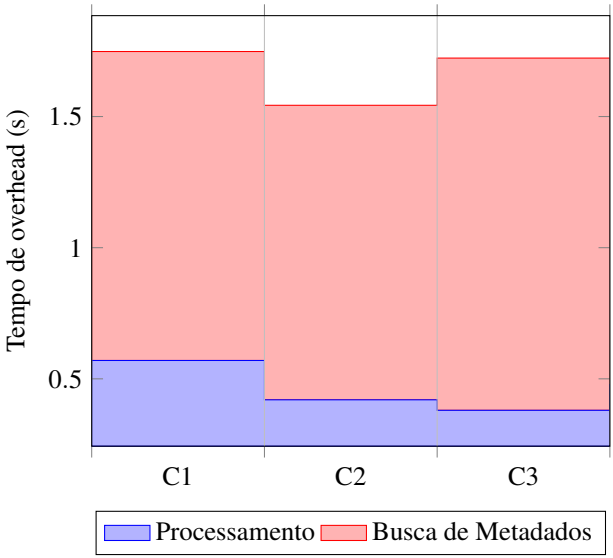

Figura 30 – Overhead da ferramenta

[1] Processamento em excesso

6 CONCLUSÃO

Tornam-se evidentes, através deste trabalho, as dificuldades de serviços em realizar mudanças na especificação de APIs uma vez que estas já possuem clientes fazendo o seu acesso. Além disso, nota-se que é possível explorar novos modelos de comunicação a fim de manter a eficiência da comunicação cliente-servidor evitando versionamento.

O esforço dedicado neste trabalho em explorar esta área culminou no desenvolvimento de um novo modelo de comunicação. Buscou-se eliminar a preocupação de clientes em estar continuamente atualizando seu código de busca a cada mudança na API e direcionar desenvolvedores de clientes à implementação de códigos de busca independentes de especificação de API.

Apesar das vantagens do modelo, existe um investimento a mais com a integração da ferramenta para que clientes e serviços possam usufruir os benefícios demonstrados nos resultados dos testes de validação. Felizmente, serviços que disponibilizam descrições de metadados da sua API apresentam uma menor barreira de entrada e já podem ser consultados utilizando a ferramenta.

Por fim, uma das principais contribuições deste trabalho é estampar os benefícios da automação na execução de consultas de dados e composição de serviços que o modelo e a ferramenta podem proporcionar. Basta que os desenvolvedores de clientes se preocupem em escrever um código de busca através de linguagens de consulta como o GraphQL e dos serviços em disponibilizarem uma completa descrição dos metadados de APIs.

6.1 TRABALHOS FUTUROS

- Realização de testes de validação para a composição de serviços.
- Criação de novos adaptadores da ferramenta para formatos de descrição de APIs. (OpenAPI, RAML, API Blueprint)
- Implementação da especificação da ferramenta em outras plataformas de desenvolvimento. (Mobile, Desktop, etc)
- Aprimoramento do algoritmo de análise de consultas.

REFERÊNCIAS

ART, A. *Tracking the Growth of the API Economy | Nordic APIs |*. 2016. Disponível em: <http://nordicapis.com/tracking-the-growth-of-the-api-economy/>.

DROETTBOOM, M. *Understanding JSON Schema ó Understanding JSON Schema 1.0 documentation*. 2015. Disponível em: <https://spacetelescope.github.io/understanding-json-schema/>.

DUVANDER, A. *9,000 APIs: Mobile Gets Serious*. 2013. Disponível em: <http://www.programmableweb.com/news/9000-apis-mobile-gets-serious/2013/04/30>.

DUVANDER, A. *JSON's Eight Year Convergence With XML*. 2013. Disponível em: <http://www.programmableweb.com/news/jsons-eight-year-convergence-xml/2013/12/26>.

FACEBOOK. *GraphQL*. 2016. Disponível em: <http://facebook.github.io/graphql/>.

FIELDING, R. T. Architectural styles and the design of network-based software architectures. *University of California, Irvine*, 2000. Disponível em: <http://dl.acm.org/citation.cfm?id=932295>.

JACKSON, W. *JSON quick syntax reference*. [S.l.: s.n.], 2016.

KNUPP, J. *Why I Hate HATEOAS*. 2016. Disponível em: <https://jeffknupp.com/blog/2014/06/03/why-i-hate-hateoas/>.

LEACH, B. *Elegant APIs with JSON Schema ó Brandur Leach*. 2014. Disponível em: <https://brandur.org/elegant-apis>.

LUCKY, M. et al. Enriching api descriptions by adding api profiles through semantic annotation. *Springer International Publishing*, p. 780–794, 2016. Disponível em: <http://link.springer.com/chapter/10.1007/978-3-319-46295-0_55>.

LYNN, K.; DORNIN, L. *Modeling RESTful APIs with JSON Hyper-Schema*. 2016. Disponível em: <https://www.iab.org/wp-content/IAB-uploads/2016/03/draft-lynn-t2trg-model-rest-apis-00.txt>.

PAUTASSO, C.; ZIMMERMANN, O.; LEYMANN, F. Restful web services vs. "big"' web services. *Proceeding of the 17th international conference on World Wide Web - WWW '08*, 2008.

PEZOA, F. et al. Foundations of json schema. *Proceedings of the 25th International Conference on World Wide Web*, p. 263–273, 2016. Disponível em: <http://dl.acm.org/citation.cfm?id=2883029>.

RICHARDSON LEONARDAMUNDSEN, M. *RESTful Web APIs*. [S.l.]: O'Reilly, 2013.

SANDOVAL, K. *Top Specification Formats for REST APIs | Nordic APIs |*. 2015. Disponível em: <http://nordicapis.com/top-specification-formats-for-rest-apis/>.

WILDERMUTH, S. *REST matters (and you need more of it)*. 2015. Disponível em: <https://www.pluralsight.com/blog/tutorials/representational-state-transfer-tips>.

ZYP, K. *JSON Schema: core definitions and terminology*. 2013. Disponível em: <http://json-schema.org/latest/json-schema-core.html>.

FUNÇÕES DA FERRAMENTA

```
import {buildSchema} from "./buildSchema"
import {deepExtendSchema} from "./deepExtendSchema"
import {deepExtendData} from "./deepExtendData"
import {wrapData} from "./wrapData"
import {wrapSchema} from "./wrapSchema"
import {unwrapAST} from "./unwrapAST"
import {simplifyAST} from "./simplifyAST"
import {reduceASTs} from "./reduceASTs"
import {fetchData} from "./fetchData"
import {transformAST} from "./transformAST"

export function composeSchema(...services) {
  return Promise.all(services.map((service) => {
    return service()
  })).then((services) => {
    return Promise.all(services.map((service) => {
      var {schema, adapter} = service

      if (adapter) {
        return adapter.buildSchema(schema)
      } else {
        return buildSchema(schema)
      }
    })).then((schemas) => {
      return Promise.all(services.map((service, index) => {
        var {wrapper} = service
        var schema = schemas[index]
        return wrapSchema(schema, wrapper)
      })).then((wrappedSchemas) => {
        return deepExtendSchema(...wrappedSchemas).then((rootSchema)
        ↪  => {
          var queryFields = rootSchema.getQueryType().getFields()

          Object.keys(queryFields).forEach((queryFieldName) => {
            var queryField = queryFields[queryFieldName]

            queryField.resolve = (parent, args, context, info) => {
              var rootAST = simplifyAST({
                "selectionSet": {
```

```
39                      "kind": "SelectionSet",
40                      "selections": info.fieldASTs
41                    }
42                  }, info)
43
44                  var asts = services.map((service, index) => {
45                    var {metadata, adapter} = service
46                    var schema = wrappedSchemas[index]
47
48                    if (adapter) {
49                      return adapter.transformAST(metadata, schema,
                          ↪  rootAST)
50                    } else {
51                      return transformAST(metadata, schema, rootAST)
52                    }
53                  })
54
55                  asts = reduceASTs(rootAST, ...asts)
56
57                  var requests = services.map((service, index) => {
58                    var {metadata, adapter, url, wrapper, fetch} = service
59                    var wrappedSchema = wrappedSchemas[index]
60                    var schema = schemas[index]
61                    var ast = unwrapAST(asts[index], wrappedSchema,
                          ↪  wrapper)
62                    var request
63
64                    if (adapter) {
65                      request = adapter.fetchData(metadata, ast, url,
                          ↪  fetch)
66                    } else {
67                      request = fetchData(metadata, ast, url, fetch)
68                    }
69
70                    return request.then((data) => {
71                      return wrapData(data, schema, wrapper)
72                    })
73                  })
74
75                  return Promise.all(requests).then((responses) => {
76                    return deepExtendData(...responses)
77                  }).then((response) => {
```

```
                return response[info.fieldName]
              })
            }
          })

          return rootSchema
        })
      })
    })
  })
}
```

```
import {buildClientSchema} from "graphql"

export function buildSchema(metadata) {
  return buildClientSchema(metadata)
}
```

```
import clone from "clone"
import {graphql, introspectionQuery, buildClientSchema} from 'graphql'

export function wrapSchema(schema, wrapper={}) {
  return graphql(schema, introspectionQuery).then((response) => {
    var schemaData = response.data
    var types = schemaData.__schema.types

    Object.keys(wrapper).forEach((typeName) => {
      var wrap = wrapper[typeName]
      var type = types.find((type) => {
        return type.name == typeName
      })

      Object.keys(wrap).forEach((wrapFieldName) => {
        var fieldName = wrap[wrapFieldName]
        var wrapField = clone(fieldpath(type.name, fieldName, types))

        type.fields = type.fields.filter((field) => {
          return field.name != wrapFieldName
```

```
          })

          wrapField.name = wrapFieldName
          type.fields.push(wrapField)
        })
      })

      return buildClientSchema(schemaData)
    })
  }

  function fieldpath(typeName, fieldName, types) {
    var fieldNames = fieldName.split(".")
    var type = types.find((type) => {
      return type.name == typeName
    })
    var field = type.fields.find((field) => {
      return field.name == fieldNames[0]
    })

    if (fieldNames.length == 1) {
      return field
    } else {
      fieldNames.shift()

      if (field.type.ofType) {
        return fieldpath(field.type.ofType.name, fieldNames.join("."),
          ↪  types)
      } else {
        return fieldpath(field.type.name, fieldNames.join("."), types)
      }
    }
  }
```

```
  import clone from 'clone'
  import {graphql, introspectionQuery, buildClientSchema} from 'graphql'

  export function deepExtendSchema(...schemas) {
    return Promise.all(schemas.map((schema) => {
      return graphql(schema, introspectionQuery).then((response) => {
```

```
7           return response.data
8         })
9       })).then((schemasData) => {
10        var rootSchemaData = {
11          "__schema": {
12            "queryType": {
13              "name": "GraphQLJayQueryType"
14            },
15            "types": []
16          }
17        }
18
19        schemasData.forEach((schemaData) => {
20          schemaData = clone(schemaData)
21
22          var queryTypeName = schemaData.__schema.queryType.name
23          schemaData.__schema.queryType.name = "GraphQLJayQueryType"
24
25          schemaData.__schema.types.forEach((type) => {
26            if (type.name == queryTypeName) {
27              type.name = "GraphQLJayQueryType"
28            }
29
30            var rootType = rootSchemaData.__schema.types.find((rootType)
              ↪  => {
31              return rootType.name == type.name
32            })
33
34            if (rootType) {
35              extendType(rootType, type)
36            } else {
37              rootSchemaData.__schema.types.push(type)
38            }
39          })
40        })
41
42        return buildClientSchema(rootSchemaData)
43      })
44    }
45
46    function extendType(rootType, type) {
47      var fields = type.fields || []
```

```
48
49      fields.forEach((field) => {
50        var rootField = rootType.fields.find((rootField) => {
51          return rootField.name == field.name
52        })
53
54        if (rootField) {
55          extendTypeField(rootField, field)
56        } else {
57          rootType.fields.push(field)
58        }
59      })
60    }
61
62    function extendTypeField(rootField, field) {
63      var args = field.args || []
64
65      args.forEach((arg) => {
66        var rootArg = rootField.args.find((rootArg) => {
67          return rootArg.name == arg.name
68        })
69
70        if (rootArg) {
71          Object.assign(rootArg, arg)
72        } else {
73          rootField.args.push(arg)
74        }
75      })
76    }
```

```
1     //
       ↪  https://github.com/mickhansen/graphql-sequelize/blob/master/src/simplifyAST.js
2
3     function deepMerge(a, b) {
4       Object.keys(b).forEach(function (key) {
5         if (['fields', 'args'].indexOf(key) !== -1) return
6
7         if (a[key] && b[key] && typeof a[key] === 'object' && typeof
           ↪  b[key] === 'object') {
8           a[key] = deepMerge(a[key], b[key])
```

```
9         } else {
10          a[key] = b[key]
11        }
12      })
13
14      if (a.fields && b.fields) {
15        a.fields = deepMerge(a.fields, b.fields)
16      } else if (a.fields || b.fields) {
17        a.fields = a.fields || b.fields
18      }
19
20      return a
21    }
22
23    function hasFragments(info) {
24      return info.fragments && Object.keys(info.fragments).length > 0
25    }
26
27    function isFragment(info, ast) {
28      return hasFragments(info) && info.fragments[ast.name.value] &&
        ↪  ast.kind !== 'FragmentDefinition'
29    }
30
31    function simplifyObjectValue(objectValue) {
32      return objectValue.fields.reduce((memo, field) => {
33        memo[field.name.value] =
34          field.value.kind === 'IntValue' ? parseInt( field.value.value,
           ↪  10 ) :
35          field.value.kind === 'FloatValue' ? parseFloat(
           ↪  field.value.value ) :
36          field.value.kind === 'ObjectValue' ? simplifyObjectValue(
           ↪  field.value ) :
37            field.value.value
38
39        return memo
40      }, {})
41    }
42
43    function simplifyValue(value, info) {
44      if (value.values) {
45        return value.values.map(value => simplifyValue(value, info))
46      }
```

```
47      if ('value' in value) {
48        return value.value
49      }
50      if (value.kind === 'ObjectValue') {
51        return simplifyObjectValue(value)
52      }
53      if (value.name) {
54        return info.variableValues[value.name.value]
55      }
56    }
57
58    export function simplifyAST(ast, info, parent) {
59      var selections
60      info = info || {}
61
62      if (ast.selectionSet) selections = ast.selectionSet.selections
63      if (Array.isArray(ast)) {
64        let simpleAST = {}
65        ast.forEach(ast => {
66          simpleAST = deepMerge(
67            simpleAST, simplifyAST(ast, info)
68          )
69        })
70
71        return simpleAST
72      }
73
74      if (isFragment(info, ast)) {
75        return simplifyAST(info.fragments[ast.name.value], info)
76      }
77
78      if (!selections) return {
79        fields: {},
80        args: {}
81      }
82
83      return selections.reduce(function (simpleAST, selection) {
84        if (selection.kind === 'FragmentSpread' || selection.kind ===
          ↪  'InlineFragment') {
85          simpleAST = deepMerge(
86            simpleAST, simplifyAST(selection, info)
87          )
```

```
88          return simpleAST
89        }
90
91        var name = selection.name.value
92          , alias = selection.alias && selection.alias.value
93          , key = alias || name
94
95        simpleAST.fields[key] = simpleAST.fields[key] || {}
96        simpleAST.fields[key] = deepMerge(
97          simpleAST.fields[key], simplifyAST(selection, info,
             ↪  simpleAST.fields[key])
98        )
99
100       if (alias) {
101         simpleAST.fields[key].key = name
102       }
103
104       simpleAST.fields[key].args = selection.arguments.reduce(function
           ↪  (args, arg) {
105         args[arg.name.value] = simplifyValue(arg.value, info)
106         return args
107       }, {})
108
109       if (parent) {
110         Object.defineProperty(simpleAST.fields[key], '$parent', { value:
             ↪  parent, enumerable: false })
111       }
112
113       return simpleAST
114     }, {
115       fields: {},
116       args: {}
117     })
118   }
```

```
1   import clone from 'clone'
2
3   export function transformAST(metadata, schema, ast) {
4     ast = clone(ast)
5
```

```
    var reduceAST = (type, fields) => {
      Object.keys(fields).forEach((fieldName) => {
        var field = fields[fieldName]
        var typeField = type.getFields()[fieldName]

        if (!typeField) {
          delete fields[fieldName]
        } else {
          reduceAST(typeField.type.ofType || typeField.type,
            ↪  field.fields)
        }
      })
    }

    reduceAST(schema.getQueryType(), ast.fields)

    return ast
  }
```

```
  import clone from 'clone'
  import keypath from 'keypath'
  import traverse from 'traverse'

  export function reduceASTs(rootAST, ...asts) {
    var rootFieldPaths = fieldPathsFor(rootAST)

    asts = asts.map((ast) => {
      return clone(ast)
    })

    clone(asts).sort((ast) => {
      return heuristic(rootFieldPaths, ast)
    }).reverse()

    asts.forEach((ast) => {
      var index = asts.indexOf(ast)
      var fieldPaths = fieldPathsFor(ast)

      asts.slice(index + 1).forEach((_ast) => {
        reduceAST(_ast, fieldPaths)
```

```
22        })
23      })
24
25      return asts
26    }
27
28    function deleteFieldPath(ast, fieldPath) {
29      var keys = fieldPath.split(".")
30      var lastKey = keys[keys.length - 1]
31      keys.pop()
32      var key = keys.join(".")
33      var value = keypath(key, ast)
34      delete value[lastKey]
35    }
36
37    function reduceAST(ast, fieldPaths) {
38      var objectFieldPaths = []
39
40      fieldPaths.forEach((fieldPath) => {
41        var field = keypath(fieldPath, ast)
42
43        if (field) {
44          if (Object.keys(field.fields).length > 0) {
45            objectFieldPaths.push(fieldPath)
46          } else {
47            deleteFieldPath(ast, fieldPath)
48          }
49        }
50      })
51
52      objectFieldPaths.reverse().forEach((objectFieldPath) => {
53        var field = keypath(objectFieldPath, ast)
54
55        if (Object.keys(field.fields).length == 0) {
56          deleteFieldPath(ast, objectFieldPath)
57        }
58      })
59    }
60
61    function heuristic(rootFieldPaths, ast) {
62      var fieldPaths = fieldPathsFor(ast)
63      var totalFields = rootFieldPaths.length
```

```
    var totalFieldsIncluded = 0
    var totalFieldsExtra = 0

    fieldPaths.forEach((fieldPath) => {
      if (rootFieldPaths.indexOf(fieldPath) >= 0) {
        totalFieldsIncluded++
      } else {
        totalFieldsExtra++
      }
    })

    return ((totalFieldsIncluded/totalFields) * (1 - (0.01 *
     ↪  totalFieldsExtra)))
  }

  function fieldPathsFor(ast) {
    var fieldPaths = []

    traverse(ast).forEach(function(value) {
      var isFields = this.path[this.path.length - 2] == "fields"

      if (isFields) {
        fieldPaths.push(this.path.join("."))
      }
    })

    return fieldPaths
  }
```

```
  import clone from 'clone'

  export function unwrapAST(ast, schema, wrapper={}) {
    ast = clone(ast)

    var unwrapFields = (type, fields) => {
      var wrap = wrapper[type.name]
      var nextFields = {}

      if (wrap) {
        Object.keys(wrap).forEach((wrapFieldName) => {
```

```
12          var wrapPath = wrap[wrapFieldName]
13
14          if (fields[wrapFieldName]) {
15            var [firstFieldName, secondFieldName] = wrapPath.split(".")
16
17            fields[firstFieldName] = {
18              fields: {
19                [secondFieldName]: fields[wrapFieldName]
20              },
21              args: []
22            }
23
24            if (firstFieldName != wrapFieldName) {
25              delete fields[wrapFieldName]
26            } else {
27              nextFields[firstFieldName] =
                  ↪  fields[firstFieldName].fields[secondFieldName].fields
28            }
29          }
30        })
31      }
32
33      Object.keys(fields).forEach((fieldName) => {
34        var field = fields[fieldName]
35
36        if (type.getFields && type.getFields()[fieldName]) {
37          var typeField = type.getFields()[fieldName]
38          unwrapFields(typeField.type.ofType || typeField.type,
              ↪  nextFields[fieldName] || field.fields)
39        }
40      })
41    }
42
43    unwrapFields(schema.getQueryType(), ast.fields)
44
45    return ast
46  }
```

```
1   import fetch from "isomorphic-fetch"
2
```

```
3   export function fetchData(metadata, ast, url, fetchFn) {
4     var query = buildQuery(ast)
5     var performFetch = fetchFn || fetch
6
7     if (query) {
8       return performFetch(url, {
9         body: JSON.stringify({
10          query
11        }),
12        headers: {
13          'Accept': 'application/json',
14          'Content-Type': 'application/json'
15        },
16        method: "POST"
17      }).then((response) => {
18        return response.json()
19      }).then((response) => {
20        return response.data
21      })
22    } else {
23      return Promise.resolve({})
24    }
25  }
26
27  function buildQuery(ast) {
28    var buildSelection = (fields) => {
29      var query = ""
30
31      Object.keys(fields).forEach((fieldName) => {
32        var field = fields[fieldName]
33
34        var str = `${fieldName}`
35
36        if (Object.keys(field.args).length > 0) {
37          str += "("
38
39          Object.keys(field.args).forEach((argName) => {
40            if (typeof field.args[argName] == "string") {
41              str += `${argName}: "${field.args[argName]}"`
42            } else {
43              str += `${argName}: ${field.args[argName]}`
44            }
```

```
45            })
46
47            str += ")"
48          }
49
50          if (Object.keys(field.fields).length > 0) {
51            query += `${str} ${buildSelection(field.fields)}`
52          } else {
53            query += `${str} `
54          }
55
56          query = query.replace("  ", " ")
57        })
58
59        if (query.trim() != "") {
60          return `{ ${query} }`
61        }
62      }
63
64      return buildSelection(ast.fields)
65    }
```

```
1   export function wrapData(data, schema, wrapper={}) {
2     var wrapFields = (type, fields) => {
3       var wrap = wrapper[type.name]
4
5       if (wrap) {
6         Object.keys(wrap).forEach((wrapFieldName) => {
7           var wrapPath = wrap[wrapFieldName]
8           var [firstFieldName, secondFieldName] = wrapPath.split(".")
9           var fieldsList = [fields];
10
11          if (Array.isArray(fields)) {
12            fieldsList = fields;
13          }
14
15          fieldsList.forEach((fields) => {
16            if (fields[firstFieldName] &&
                ↪  fields[firstFieldName][secondFieldName]) {
```

```
17              fields[wrapFieldName] =
                 ↪  fields[firstFieldName][secondFieldName]
18
19              if (wrapFieldName != firstFieldName) {
20                delete fields[firstFieldName]
21              }
22
23              var typeField =
                 ↪  type.getFields()[firstFieldName].type.getFields()[secondFieldName]
24
25              if (typeof fields[wrapFieldName] == "object") {
26                wrapFields(typeField.type.ofType || typeField.type,
                   ↪  fields[wrapFieldName])
27              }
28            }
29          })
30        })
31      }
32
33      Object.keys(fields).forEach((fieldName) => {
34        var field = fields[fieldName]
35
36        if (type.getFields && type.getFields()[fieldName]) {
37          var typeField = type.getFields()[fieldName]
38          wrapFields(typeField.type.ofType || typeField.type, field)
39        }
40      })
41    }
42
43    wrapFields(schema.getQueryType(), data)
44
45    return data
46  }
```

```
1   import deepAssign from "deep-assign"
2
3   export function deepExtendData(...data) {
4     return deepAssign(...data)
5   }
```

FUNÇÕES DO ADAPTADOR

```
import $RefParser from 'json-schema-ref-parser'
import {buildClientSchema as graphQLBuildClientSchema} from 'graphql'
import uuid from "uuid"
import traverse from "traverse"
import URITemplateParser from 'uri-template'

var typePropertyMap = {}

export function buildSchema(hyperSchema) {
  return $RefParser.dereference(hyperSchema).then((hyperSchema) => {
    var customTypes = customTypesFromHyperSchema(hyperSchema)
    var queryType = queryTypeFromHyperSchema(hyperSchema, customTypes)

    return graphQLBuildClientSchema({
      "__schema": {
        "queryType": {
          "name": queryType.name
        },
        "types": [
          queryType,
          ...customTypes
        ]
      }
    })
  })
}

function customTypesFromHyperSchema(hyperSchema) {
  var types = []

  traverse(hyperSchema).forEach(function(value) {
    if (value && value.properties && value.title != hyperSchema.title
     ↪  && this.key != "schema") {
      var property = value

      var found = types.find((type) => {
        return type.name == property.title
      })

```

```
39          if (!found) {
40            var name = property.title || `__${uuid.v4().split("-")[0]}`
41            var type = {
42              "name": name,
43              "kind": "OBJECT",
44              "fields": [],
45              "interfaces": []
46            }
47
48            typePropertyMap[name] = property
49            types.push(type)
50          }
51        }
52      })
53
54      types.forEach((type) => {
55        var property = typePropertyMap[type.name]
56
57        type.fields =
          ↪  Object.keys(property.properties).map((fieldPropertyName) => {
58          var fieldProperty = property.properties[fieldPropertyName]
59          var links = property.links || []
60
61          var links = links.filter((link) => {
62            return link.rel == fieldPropertyName
63          })
64
65          return parseField(fieldPropertyName, fieldProperty, links,
            ↪  {types})
66        })
67      })
68
69      return types
70    }
71
72    function queryTypeFromHyperSchema(hyperSchema, types) {
73      var queryType = {
74        "name": "Query",
75        "kind": "OBJECT",
76        "fields": [],
77        "interfaces": []
78      }
```

```
79
80      Object.keys(hyperSchema.properties).forEach((propertyName) => {
81        var property = hyperSchema.properties[propertyName]
82        var links = hyperSchema.links || []
83
84        var links = links.filter((link) => {
85          return link.rel == propertyName
86        })
87
88        queryType.fields.push(parseField(propertyName, property, links,
          ↪  {types}))
89      })
90
91      return queryType
92    }
93
94    function parseField(name, property, links, {types}) {
95      var args = []
96
97      links.forEach((link) => {
98        var uriTemplate = URITemplateParser.parse(link.href)
99
100       uriTemplate.expressions.forEach((expression) => {
101         expression.params.forEach((param) => {
102           if (link.schema) {
103             var argProperties = link.schema.properties
104             var argProperty = argProperties[param.name]
105
106             args.push({
107               name: param.name,
108               type: convertType(argProperty, types)
109             })
110           }
111         })
112       })
113     })
114
115     if (links.length > 0) {
116       property = links[0].targetSchema
117     }
118
119     var type = convertType(property, types)
```

```
120
121       return {
122         "name": name,
123         "args": args,
124         "type": type
125       }
126     }
127
128     function convertType(property, types) {
129       if (property.type == "array") {
130         var type = types.find((type) => {
131           return property.items == typePropertyMap[type.name]
132         })
133
134         if (type) {
135           return {
136             "kind": "LIST",
137             "ofType": {
138               "kind": type.kind,
139               "name": type.name
140             }
141           }
142         } else {
143           return {
144             "kind": "LIST",
145             "ofType": {
146               "kind": "SCALAR",
147               "name": "String"
148             }
149           }
150         }
151       } else if (property.type == "boolean") {
152         return {
153           "kind": "SCALAR",
154           "name": "Boolean"
155         }
156       } else if (property.type == "integer") {
157         return {
158           "kind": "SCALAR",
159           "name": "Int"
160         }
161       } else if (property.type == "number") {
```

```
      return {
        "kind": "SCALAR",
        "name": "Float"
      }
    } else if (property.type == "string") {
      return {
        "kind": "SCALAR",
        "name": "String"
      }
    } else if (property.type == "object") {
      var type = types.find((type) => {
        return property == typePropertyMap[type.name]
      })

      return {
        "kind": "OBJECT",
        "name": type.name
      }
    } else {
      return {
        "kind": "SCALAR",
        "name": "String"
      }
    }
  }
```

```
import {transformAST as graphQLJayTransformAST} from "graphql-jay"

export function transformAST(hyperSchema, schema, ast) {
  var ast = graphQLJayTransformAST(hyperSchema, schema, ast)

  // TODO: add over-fetching attrs

  return ast
}
```

```
import fetch from "isomorphic-fetch"
import URITemplateParser from 'uri-template'
```

```
3   import Bluebird from 'bluebird'
4   import deepEqual from 'deep-equal'
5
6   export function fetchData(hyperSchema, ast, url, fetchFn) {
7     return resolveFields(ast.fields, hyperSchema, {}, url,
      ↪  fetchFn).then((data) => {
8       return data
9     })
10  }
11
12  function resolveFields(fields, property, parentData, url, fetchFn) {
13    var data = {}
14    var requests = []
15
16    if (Object.keys(fields).length == 0) {
17      return Promise.resolve(data)
18    }
19
20    Object.keys(fields).forEach((astFieldName) => {
21      var astField = fields[astFieldName]
22      var links = property.links || []
23      var link = links.find((link) => {
24        return link.rel == astFieldName
25      })
26
27      var relLink = links.find((_link) => {
28        return _link.linkRel && _link.linkRel.id == link.id &&
          ↪  deepEqual(_link.linkRel.values, astField.args)
29      })
30
31      if (relLink) {
32        link = relLink
33      }
34
35      if (link) {
36        requests.push(resolveLink(link, astField.args, parentData, url,
          ↪  fetchFn).then((fieldData) => {
37          data[astFieldName] = fieldData
38
39          var targetProperty = link.targetSchema
40
41          if (link.targetSchema.type == "array") {
```

```
42              targetProperty = link.targetSchema.items
43            }
44
45            if (Array.isArray(fieldData)) {
46              return batch(fieldData, (fieldItemData, index) => {
47                return resolveFields(astField.fields, targetProperty,
                    ↪  fieldItemData, url, fetchFn).then((resolvedFieldData)
                    ↪  => {
48                  fieldData[index] = Object.assign(fieldItemData,
                      ↪  resolvedFieldData)
49                })
50              })
51            } else {
52              return resolveFields(astField.fields, targetProperty,
                  ↪  fieldData, url, fetchFn).then((resolvedFieldData) => {
53                data[astFieldName] = Object.assign(fieldData,
                    ↪  resolvedFieldData)
54              })
55            }
56          }))
57        } else {
58          var fieldProperty = property.properties[astFieldName]
59          var fieldData = parentData[astFieldName]
60
61          if (fieldProperty.type == "array") {
62            requests.push(batch(fieldData, (fieldItemData, index) => {
63              return resolveFields(astField.fields, fieldProperty.items,
                    ↪  fieldItemData, url, fetchFn).then((resolvedFieldData)
                    ↪  => {
64                fieldData[index] = Object.assign(fieldItemData,
                    ↪  resolvedFieldData)
65              })
66            }))
67          } else if (fieldProperty.type == "object") {
68            requests.push(resolveFields(astField.fields, fieldProperty,
                  ↪  fieldData, url, fetchFn).then((resolvedFieldData) => {
69              data[astFieldName] = Object.assign(fieldData,
                    ↪  resolvedFieldData)
70            }))
71          }
72        }
73      })
```

```
74
75      return batch(requests).then(() => {
76        return data
77      })
78    }
79
80    function resolveLink(link, astFieldArgs, parentData, url, fetchFn) {
81      var performFetch = fetchFn || fetch
82      var uriTemplate = URITemplateParser.parse(link.href)
83      var context = Object.assign({}, astFieldArgs, parentData)
84
85      if (link.each) {
86        return batch(context[link.each], (item) => {
87          var uri = uriTemplate.expand(Object.assign({}, context, {item}))
88
89          if (!/^[a-zA-Z]+:\/\//.test(uri)) {
90            uri = `${url}${uri}`
91          }
92
93          return performFetch(uri, {
94            headers: {
95              'Accept': 'application/json',
96              'Content-Type': 'application/json'
97            }
98          }).then((response) => {
99            return response.json()
100         })
101       })
102     } else {
103       var uri = uriTemplate.expand(context)
104
105       if (!/^[a-zA-Z]+:\/\//.test(uri)) {
106         uri = `${url}${uri}`
107       }
108
109       return performFetch(uri, {
110         headers: {
111           'Accept': 'application/json',
112           'Content-Type': 'application/json'
113         }
114       }).then((response) => {
115         return response.json()
```

```
116       })
117     }
118   }
119
120   function batch(items, method) {
121     var result = []
122
123     return Bluebird.each(items, (item, index) => {
124       if (method) {
125         return method(item, index).then((response) => {
126           result.push(response)
127         })
128       } else {
129         result.push(item)
130       }
131     }).then(() => {
132       return result
133     })
134   }
```

Um Modelo de Comunicação para Automação na Execução de Consultas de Dados sobre APIs Web

Mateus Maso[1], **Frank Siqueira**[2]

[1]Departamento de Informática e Estatística (INE)
Universidade Federal de Santa Catarina (UFSC) – Florianópolis, SC – Brazil

mateus.maso@grad.ufsc.br, frank.siqueira@ufsc.br

Abstract. *In order to maintain client-server communication efficiency without the need of versioning Web APIs, services have come across problems while performing changes on their interface access specification due to the coupling caused by clients on the implementation of its fetching code. Seeking to develop a solution for the problem, this project conducts a study on the usage of GraphQL language and API description formats to propose a client-server communication model through automation in the execution of data queries. As a result, a tool foresaw by the proposed model is developed to validate its applicability and guide client-side developers to the implementation of data fetching code independent of API specification.*

Resumo. *A fim de manter a eficiência da comunicação cliente-servidor sem a necessidade do versionamento de APIs Web, serviços têm encontrado dificuldades em realizar mudanças na especificação de sua interface de acesso devido ao acoplamento causado por clientes na implementação em seu código de busca. No intuito de desenvolver uma solução para o problema encontrado, este trabalho realiza um estudo sobre o uso da linguagem GraphQL e formatos de descrição de API para propor um modelo de comunicação cliente-servidor através da automação na execução de consultas de dados. Como resultado, é desenvolvida uma ferramenta prevista pelo modelo proposto para validar sua aplicabilidade e direcionar desenvolvedores de clientes à implementação de um código de busca independente de especificação de API.*

1. Introdução

Em 2005, ocorreu uma grande transição no modelo de comunicação entre aplicações distribuídas, onde estas passaram a utilizar amplamente o protocolo HTTP e o modelo cliente-servidor para a troca de informações na World Wide Web. Hoje, empresas como Facebook e Netflix mostram que construir APIs Web é, não apenas essencial para entrar rápido no mercado de plataformas emergentes, como também um novo meio de agregar valor em seu próprio modelo de negócio e oferecer uma melhor experiência a seus usuários. [Duvander 2013, Art 2016]

Contudo, após anos de sua popularização, serviços têm mostrado dificuldades em realizar mudanças em suas APIs Web sem comprometer a comunicação de clientes. Isso porque clientes continuam implementando seu código de busca através de chamadas diretamente na API, ocasionando um acoplamento da especificação muitas vezes indesejado pelos serviços.

Com o objetivo de resolver o problema de acoplamento, este trabalho propõe um novo modelo de comunicação cliente-servidor através da automação na execução de consultas de dados sobre APIs Web. O modelo visa direcionar desenvolvedores de clientes à implementação de códigos de busca independente de especificação de API, resultando no desenvolvimento de clientes tolerantes às mudanças na especificação. Ainda, prevê o desenvolvimento de uma ferramenta que funciona como intermediadora na comunicação e automação de consultas, permitindo otimização de requisições e composição de serviços para consulta de dados.

2. Desenvolvimento

2.1. Planejamento de Projeto

Para que clientes possam trabalhar com dados remotos, é preciso que haja um meio de buscá-los antes de executar qualquer lógica que dependa deles. Para isso, a solução comumente adotada é a implementação de um código de busca para acesso remoto de dados em APIs de serviços[1]. Contudo, ao passar do tempo, fez-se necessário estabelecer um contrato de acesso entre o cliente e a interface do serviço. Isso porque a atual implementação do código de busca por clientes não prevê mudanças na especificação da API. Este contrato é, portanto, representado no modelo de comunicação da figura 1 através de chamadas entre o código de busca do cliente e a API do serviço.

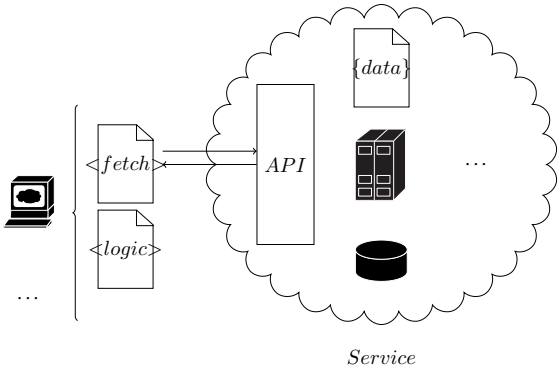

Figura 1. Modelo de comunicação entre cliente e serviço

O modelo de comunicação representado pode não revelar problemas para serviços com pouca demanda de acesso e diversidade de consultas. Contudo, ao longo do tempo e com o aumento na quantidade de contratos, alterações na especificação como, por exemplo, as de fluxo de dados[2] podem se tornar um desafio.

Mudanças no fluxo de dados pela API são inevitáveis em aplicações distribuídas. Elas ocorrem para abraçar a constante transformação por clientes na execução de consul-

[1]Infraestrutura distribuída (servidores, banco de dados, etc) que respondem a pedidos de operações oriundas de clientes em forma de requisições de API.

[2]Fluxo de acesso por clientes para consulta de dados sobre APIs de serviços.

tas de dados. Com isso, busca-se manter uma comunicação eficiente através da redução no número de chamadas executadas na API e do tamanho de dados transmitidos pela rede. Essas mudanças podem ser desde uma simples alteração no nome de um método ou número argumentos, até as mais complexas, como dados de retorno e estilo de arquitetura. A figura 2 ilustra o fluxo de dados entre um cliente e um serviço.

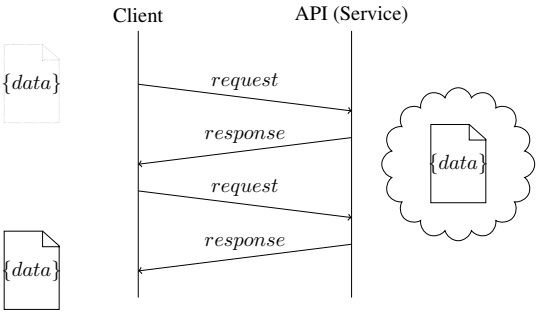

Figura 2. Fluxo de dados entre cliente e API

O impacto causado em clientes por mudanças no fluxo de dados de uma API felizmente é previsível, pois afeta diretamente, em sua maioria, o código de busca. Por outro lado, mudanças como a alteração de campos nas estruturas de dados, como renomear um campo ”nome”para ”nome_completo”, são de um grau de complexidade maior, pois pode não ter impacto no código de busca e sim na lógica da aplicação, onde é bem mais difícil de depurar erros.

2.2. Proposta de Modelo

Um novo modelo é proposto com o intuito de melhorar a comunicação cliente-servidor apresentada. Observado na figura 3, este prevê a criação de uma ferramenta no cliente para a intermediação da comunicação entre o código de busca e a API de serviços. Além disso, há a necessidade de reimplementação do código de busca para uma nova linguagem de consulta que seja interpretada pelo intermediador. Da mesma forma, soma-se a criação de um arquivo no serviço para descrição de metadados da API e outro no cliente para configuração, ambos essenciais para o funcionamento da ferramenta.

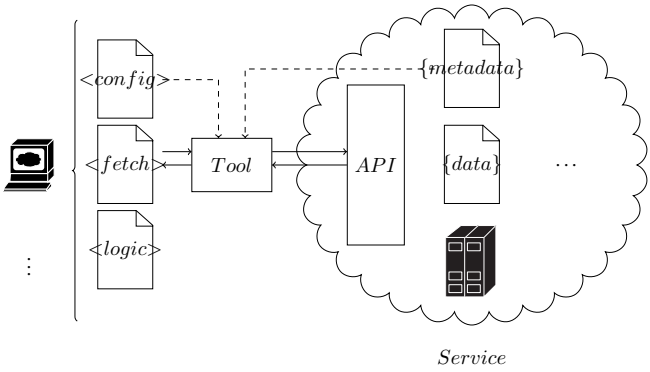

Figura 3. Modelo proposto para evitar contrato de comunicação

A proposta de eliminação de contrato visa também automatizar o acesso de clientes em APIs. Através da implementação de algoritmos na ferramenta, escolhe-se o caminho de acesso de melhor custo-benefício em relação aos serviços disponíveis para o cliente. O modelo proporciona, além disso, um ambiente escalável para consulta de dados através da composição de serviços, visto na figura 4.

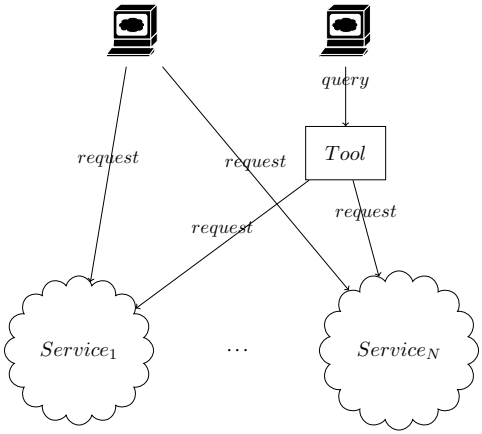

Figura 4. Composição de serviços através da ferramenta

2.3. Especificação da Ferramenta

Visando a aceitação da ferramenta prevista pelo modelo em ambientes de desenvolvimento de clientes, foi preciso pensar em uma especificação que apresentasse uma interface de baixa curva de aprendizagem, um fluxo de execução replicável e agnóstico à

plataforma, além do baixo impacto na base de código de clientes. Por conseguinte, a ferramenta proposta foi desenvolvida pensando na reutilização de tecnologias promissoras ou bem consolidadas no mercado de desenvolvimento.

Com a sucessão de estudos, a escolha foi em utilizar a tecnologia GraphQL como principal biblioteca para ajudar com intermediação da comunicação do modelo proposto. Ao invés da tecnologia Falcor (apontada durante o estudo), GraphQL foi o único que apresentou uma solução robusta que permitisse o acesso de APIs Web por clientes através de consultas em seu código de busca. Da mesma forma, buscou-se trabalhar com formatos abertos de descrição de metadados de APIs, como por exemplo o JSON Hyper-Schema. Através de adaptadores, a ferramenta permite acelerar a integração de serviços que já oferecem metadados em formatos suportados.

2.4. Implementação da Ferramenta

A fim de aplicar a especificação da ferramenta abordada no planejamento de projeto, foi proposta a implementação de onze funções, divididas em três categorias, para o uso de clientes na plataforma Web. Cada uma dessas funções levantadas resolve um problema com o intuito de chegar ao objetivo de representar os dois fluxos de execução. Dentro do conjunto de funções, quatro delas são responsáveis por criar o intermediador; três funções analisam as consultas através do formato AST; e as outras quatro transformam as consultas em requisições para API de serviços. Somente quatro são públicas e expostas para o cliente, sendo uma delas a principal para criação do esquema de consulta, e as outras três para serem sobrescritas por adaptadores.

3. Validação

Com o objetivo de validar o modelo proposto, é realizada uma pesquisa comparativa com foco no impacto da relação do uso da ferramenta implementada e mudanças no fluxo de dados sobre APIs Web. Para isso, é desenvolvido um ambiente de validação onde dois clientes, um com a ferramenta e o outro não, realizam três consultas de dados sobre uma API REST. Por fim, é aplicada uma série de quatro mudanças no fluxo de dados da API e coletados os dados para análise do impacto causado no código de busca desses clientes.

3.1. Resultados

Inicialmente, observa-se na figura 5 que o cliente 1 (sem o uso da ferramenta) não apresenta um bom índice de acerto das respostas. Dentre as quatro mudanças, obteve apenas 58% de acerto, sendo a C3 a única mudança em que conseguiu responder corretamente todas as perguntas, pois não houve mudança nas URIs que estava utilizando. Um resultado esperado, significando que houve quebra de contrato e impacto negativo no código de busca.

Em contrapartida, o cliente 2 (com o uso da ferramenta) apresenta um resultado no índice de acerto da figura 6 36.61% superior ao cliente 1. Com um total de 91,5% de acerto, apenas não completou com 100% de acerto pois a mudança C4 não permitiu que fosse possível acessar todos os dados necessários para a responder da pergunta Q2. Isso representa que a ferramenta foi capaz, através do intermediador, de evitar a criação de contrato e causar impacto negativo ao código de busca.

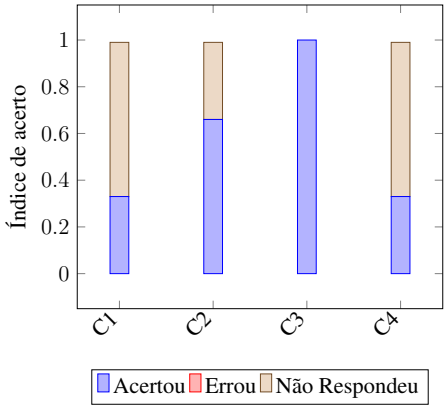

Figura 5. Índice de acerto sem o uso da ferramenta

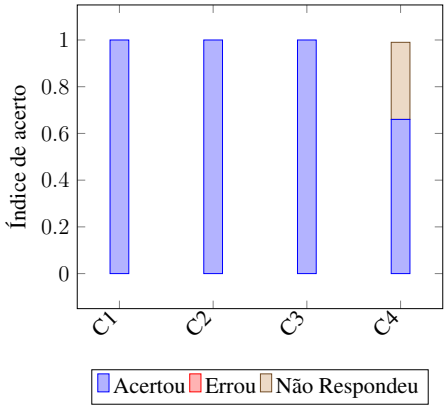

Figura 6. Índice de acerto com o uso da ferramenta

Ao analisar a mudança C3 isoladamente, onde ambos conseguem responder com 100% de acerto, percebe-se nas figuras 7 e 8 que o cliente 2 consegue realizar uma consulta com melhor desempenho (menor número de requisições e tamanho de dados) que o cliente 1. Isso porque, após a mudança e sem alterar o código de busca de dados, a ferramenta consegue remapear as requisições geradas pelas consultas GraphQL graças ao algoritmo que automatiza as chamadas à API.

Outro ponto importante é que, após a mudança C3 e a atualização dos metadados no cliente, o algoritmo da ferramenta percebe que há a possibilidade de realizar menos requisições em busca dos dados para responder as perguntas. Isso resulta em uma redução de aproximadamente 54% dos acessos entre o cliente 1 e cliente 2.

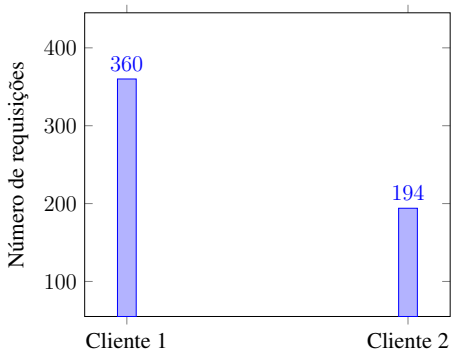

Figura 7. Comparação no número de requisições C3

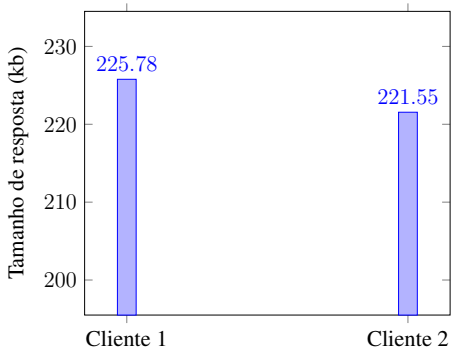

Figura 8. Comparação no tamanho de resposta C3

Apesar dos ganhos de performance e desenvolvimento através do uso da ferramenta vistos anteriormente, percebe-se na figura 9 um outro cenário que indica o lado negativo do seu uso causado pelo tempo de *overhead*[3]. Em média, a ferramenta atrasou em 1.7 segundos a execução na consulta de dados do cliente, sendo o tempo de busca de metadados responsável por somar mais da metade deste atraso inicial.

[3]Processamento em excesso

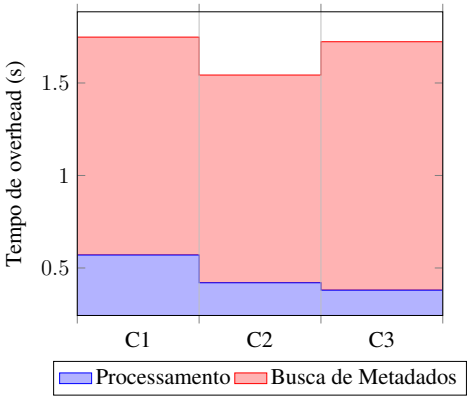

Figura 9. Overhead da ferramenta

4. Conclusão

Tornam-se evidentes, através deste trabalho, as dificuldades de serviços em realizar mudanças na especificação de APIs uma vez que estas já possuem clientes fazendo o seu acesso. Além disso, nota-se que é possível explorar novos modelos de comunicação a fim de manter a eficiência da comunicação cliente-servidor evitando versionamento.

O esforço dedicado neste trabalho em explorar esta área culminou no desenvolvimento de um novo modelo de comunicação. Buscou-se eliminar a preocupação de clientes em estar continuamente atualizando seu código de busca a cada mudança na API e direcionar desenvolvedores de clientes à implementação de códigos de busca independentes de especificação de API.

Apesar das vantagens do modelo, existe um investimento a mais com a integração da ferramenta para que clientes e serviços possam usufruir os benefícios demonstrados nos resultados dos testes de validação. Felizmente, serviços que disponibilizam descrições de metadados da sua API apresentam uma menor barreira de entrada e já podem ser consultados utilizando a ferramenta.

Por fim, uma das principais contribuições deste trabalho é estampar os benefícios da automação na execução de consultas de dados e composição de serviços que o modelo e a ferramenta podem proporcionar. Basta que os desenvolvedores de clientes se preocupem em escrever um código de busca através de linguagens de consulta como o GraphQL e dos serviços em disponibilizarem uma completa descrição dos metadados de APIs.

Referências

Art, A. (2016). Tracking the growth of the api economy — nordic apis —.

Duvander, A. (2013). Json's eight year convergence with xml.