UNIVERSIDADE DO ESTADO DO RIO GRANDE DO NORTE - UERN
FACULDADE DE CIÊNCIAS EXATAS E NATURAIS – FANAT
DEPARTAMENTO DE INFORMÁTICA – DI
CURSO DE CIÊNCIA DA COMPUTAÇÃO

ÁLVARO GABRIEL GOMES DE OLIVEIRA

**Construção de aplicações distribuídas utilizando-se de APIs REST**

MOSSORÓ - RN

2018

ÁLVARO GABRIEL GOMES DE OLIVEIRA

# Construção de aplicações distribuídas utilizando-se de APIs REST

Monografia apresentada à Universidade do Estado do Rio Grande do Norte como um dos pré-requisitos para obtenção do grau de bacharel em Ciência da Computação, sob orientação do Prof. Me. Antônio Oliveira Filho.

MOSSORÓ - RN

2018

**Catalogação da Publicação na Fonte.**
**Universidade do Estado do Rio Grande do Norte.**

O48c Oliveira, Álvaro Gabriel Gomes de
Construção de aplicações distribuídas utilizando-se de APIs REST. / Álvaro Gabriel Gomes de Oliveira. - Mossoró, 2018.
64p.

Orientador(a): Prof. Me. Antônio Oliveira Filho.
Monografia (Graduação em Ciência da Computação). Universidade do Estado do Rio Grande do Norte.

1. Estilo Arquitetural. 2. REST. 3. Internet. I. Oliveira Filho, Antônio. II. Universidade do Estado do Rio Grande do Norte. III. Título.

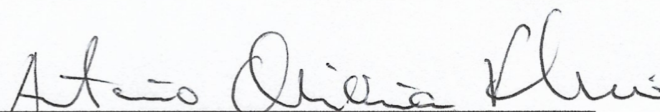
Álvaro Gabriel Gomes De Oliveira

CONSTRUÇÃO DE APLICAÇÕES DISTRIBUÍDAS UTILIZANDO-SE DE APIS REST

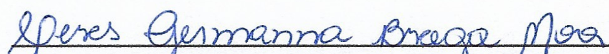
Monografia apresentada como pré-requisito para a obtenção do título de Bacharel em Ciência da Computação da Universidade do Estado do Rio Grande do Norte – UERN, submetida à aprovação da banca examinadora composta pelos seguintes membros:

Aprovada em: 23/05/2018

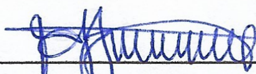
Banca Examinadora



Prof. Me. ANTÔNIO OLIVEIRA FILHO
Universidade do Estado do Rio Grande do Norte - UERN



Profa. Ma. CERES GERMANNA BRAGA MORAIS
Universidade do Estado do Rio Grande do Norte - UERN



Prof. Dr. FRANCISCO CHAGAS DE LIMA JÚNIOR
Universidade do Estado do Rio Grande do Norte - UERN

# Agradecimentos

Agradeço a todos os meus familiares, principalmente os meus pais e meu irmão, pelo apoio incondicional nessa longa caminhada.

Aos amigos do PETCC e também de toda graduação, que foram peça fundamental de um experiencia única de vida.

Aos professores do Departamento, em especial aos tutores do PETCC e o meu orientador, por passarem suas experiencias e conselhos.

*"Alegria de viver. Um dom que se conquista,
palmo a palmo. Fruto amadurecido ao calor do
entusiasmo, na árvore do dever cumprido, da
tarefa realizada, do sorriso partilhado."*
*Roque Schneider.*

# Resumo

Com a explosão da internet, as aplicações tiveram que se adaptar a uma nova realidade de comunicação. Surgiu uma necessidade iminente que as mesmas parassem de executar em um ambiente fechado e proporcionasse uma distribuição de conteúdo entre elas mesmas. Para proporcionar que essa distribuição ou comunicação ocorra entre as aplicações, requer que as mesmas sejam compostas em camadas, ou seja, tornem-se aplicações distribuídas. Criar uma aplicação distribuída hoje em dia é uma tarefa muito importante, pois com a facilidade de acesso promovida com a difusão da internet, as aplicações precisam além de se comunicarem entre si, ter a maior disponibilidade e confiabilidade possível. Portanto foi proposto para esse trabalho, a partir dos requisitos de uma aplicação, que possui fundamentos bem complexos ideal para ser uma aplicação distribuída, a criação de uma API, que se baseia no estilo arquitetural REST, para proporcionar a criação de uma aplicação distribuída. Uma API REST pode promover uma distribuição de conteúdos, de forma simples, para as mais variadas aplicações clientes, desenvolvida em qualquer linguagem. Foi implementando então, uma API em Node.js que interage com dois bancos de dados NoSQL, para implementar os requisitos da aplicação e provar que de fato a utilização de uma API, no estilo arquitetural REST, fornece características para a construção de aplicações distribuídas.

**Palavras-chave**: Estilo Arquitetural, REST, Internet.

# Abstract

With the explosion of the internet, applications had to adapt to a new reality. There was an imminent need to stop running in a closed environment and provide a distribution of content among themselves. To provide that such distribution or communication to occur between applications, requires that they are composed in layers, i.e. become distributed applications. Create a distributed application nowadays is a very important task, because with the ease of access with the diffusion of the internet applications they need in addition to communicate with each other, have the highest availability and reliability possible. So was proposed for this work, from the requirements of an application, which has very complex grounds ideal for a distributed application, creating an API, that is based on REST architectural style, to provide for the establishment of a distributed application. A REST API can promote content distribution, simply, for the most varied applications customers, developed in any language. Has been implementing so an API in Node.js that interacts with two NoSQL databases, to implement the requirements of the application and prove that in fact the use of an API, the REST architectural style, provides features for building applications distributed.

**Keywords**: Architectural Style, REST, Internet.

# Lista de ilustrações

Figura 1 – Representação do HATEOAS . . . . . . . . . . . . . . . . . . . . . . . . 21
Figura 2 – Níveis para a glória do REST . . . . . . . . . . . . . . . . . . . . . . . . 22
Figura 3 – Modelos de Dados . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
Figura 4 – Consulta AQL . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27
Figura 5 – *Token* JWT . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30
Figura 6 – Criação de uma rota na especificação *OpenAPI* . . . . . . . . . . . . . 32
Figura 7 – *Swagger UI* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33
Figura 8 – Diagrama de Classe . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
Figura 9 – Fluxograma para ilustrar os passos para o envio de respostas . . . . . . 39
Figura 10 – Balanceamento de Carga . . . . . . . . . . . . . . . . . . . . . . . . . 41
Figura 11 – Requisição ruim . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 44
Figura 12 – Fornecendo diálogo para usuários na região de Mossoró . . . . . . . . 46
Figura 13 – Fornecendo diálogo para usuários na região de Natal . . . . . . . . . . 46
Figura 14 – Questionário criado com sucesso . . . . . . . . . . . . . . . . . . . . . 47
Figura 15 – Documentação da API . . . . . . . . . . . . . . . . . . . . . . . . . . 48
Figura 16 – Interface da aplicação cliente . . . . . . . . . . . . . . . . . . . . . . . 49
Figura 17 – Acessar rota sem *token* . . . . . . . . . . . . . . . . . . . . . . . . . . 50
Figura 18 – Rotas com privilégios especiais . . . . . . . . . . . . . . . . . . . . . . 50
Figura 19 – Requisição sem informações no Redis . . . . . . . . . . . . . . . . . . . 51
Figura 20 – Requisição com informações no Redis . . . . . . . . . . . . . . . . . . . 51
Figura 21 – Visão geral da aplicação . . . . . . . . . . . . . . . . . . . . . . . . . . 52

# Lista de tabelas

Tabela 1 – Categorias dos códigos de estado do HTTP . . . . . . . . . . . . . . . . 18
Tabela 2 – Modelo RPC POX . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22
Tabela 3 – Modelo REST . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
Tabela 4 – Detalhes da rota para criar um contrato . . . . . . . . . . . . . . . . . . 44
Tabela 5 – Detalhes da rota de fornecimento de diálogos . . . . . . . . . . . . . . . 45
Tabela 6 – Detalhes da rota para criar um questionário . . . . . . . . . . . . . . . . 47

# Lista de abreviaturas e siglas

| | |
|---|---|
| ACID | Atomicity, Consistency, Isolation, Durability |
| API | Application Programming Interface |
| AQL | Arango Query Language |
| BASE | Basically Available, Soft state, Eventual consistency |
| HATEOAS | Hypermedia As The Engine Of Application State |
| HTML | Hypertext Markup Language |
| HTTP | Hypertext Transfer Protocol |
| JSON | JavaScript Object Notation |
| JWT | Json Web Token |
| NOSQL | Not Only SQL |
| NPM | Node Package Manager |
| REST | Representational State Transfer |
| SOAP | Simple Object Access Protocol |
| SQL | Structured Query Language |
| URI | Uniform Resource Identifier |
| VPS | Virtual Private Server |
| XML | eXtensible Markup Language |
| W3C | World Wide Web Consortium |

# Sumário


**1 INTRODUÇÃO** . . . . . . . . . . **13**

**2 REFERENCIAL TEÓRICO** . . . . . . . . . . **16**

**2.1 *Web Services*** . . . . . . . . . . **16**

2.1.1 Protocolo HTTP . . . . . . . . . . 17

2.1.2 REST . . . . . . . . . . 18

2.1.3 Modelo de Maturidade de Richardson . . . . . . . . . . 21

**2.2 NoSQL** . . . . . . . . . . **23**

2.2.1 *ArangoDB* . . . . . . . . . . 25

2.2.2 Redis . . . . . . . . . . 27

**2.3 Node.JS** . . . . . . . . . . **28**

2.3.1 *Express* . . . . . . . . . . 29

**2.4 JWT** . . . . . . . . . . **30**

**2.5 *OpenAPI*** . . . . . . . . . . **31**

**3 *LANGUAGE ADVISER*** . . . . . . . . . . **34**

**3.1 Visão Geral** . . . . . . . . . . **34**

**3.2 Implementação** . . . . . . . . . . **35**

3.2.1 O Modelo . . . . . . . . . . 35

3.2.2 API RESTful *Language Adviser* . . . . . . . . . . 37

3.2.3 Bancos de Dados . . . . . . . . . . 37

3.2.4 Autenticação . . . . . . . . . . 39

3.2.5 Escalabilidade . . . . . . . . . . 40

3.2.6 Geocodificação . . . . . . . . . . 41

**4 PROVA DE CONCEITO** . . . . . . . . . . **43**

**4.1 Implantando a API** . . . . . . . . . . **43**

**4.2 Rotas** . . . . . . . . . . **43**

4.2.1 Criar Contrato . . . . . . . . . . 43

4.2.2 Fornecer Diálogo . . . . . . . . . . 44

4.2.3 Criar Questionário . . . . . . . . . . 46

**4.3 Documentação** . . . . . . . . . . **47**

**4.4 Aplicação Cliente** . . . . . . . . . . **48**

**4.5 Resultados** . . . . . . . . . . **49**

**5 CONSIDERAÇÕES FINAIS** . . . . . . . . . . **53**

REFERÊNCIAS . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 55

APÊNDICE A – RECURSOS E ROTAS . . . . . . . . . . . . . . . . 58

# 1 Introdução

A necessidade de integração entre sistemas criados em diferentes linguagens é encontrada cada vez mais em grandes empresas, que necessitam da distribuição de conteúdos e de interoperabilidade. A fim de sanar questões como estas, a tecnologia dos *web services* foi criada, permitindo assim, disponibilizar formas de integrar sistemas distintos, modularizar serviços e capacitar a integração e o consumo de informações. Essa tecnologia, tradicionalmente, utiliza o protocolo *Hypertext Transfer Protocol* (HTTP) para a comunicação e troca de mensagens. Podendo então adotar um padrão que se utiliza de 100% que o protocolo HTTP oferece, nasce assim um dos padrões mais utilizado atualmente, o *Representational State Transfer* (REST) (OLIVEIRA, 2015).

A programação de *web services* de *Application Programming Interface* (API) que aderem ao estilo arquitetural REST, e de aplicações consumidoras deste tipo de serviços é atualmente muito popular. Por exemplo, aplicações como o Twitter, Instagram, Youtube e Uber, fornecem acesso às suas aplicações clientes através deste tipo de APIs (FERREIRA et al., 2017).

O REST surge atualmente como uma forma rápida, eficiente e simples de solucionar problemas de comunicações entre diferentes sistemas construídos, utilizando diferentes linguagens de programação, além de permitir a construção de uma aplicação em camadas, para tornar a mesma uma aplicação distribuída. Com a utilização do estilo arquitetural REST, a criação de uma API se torna mais simples do que se utilizando de outras tecnologias como o *Simple Object Access Protocol* (SOAP) (PULUCENO, 2012). Um *web service* é uma API que funciona na internet, que possui um conjunto de rotinas e de padrões de programação com o objetivo de integrar distintos *softwares*, possibilitando a troca de informações entre os mesmos (CIRIACO, 2009).

Nem toda API é um *web service*, mas todo *web service* é uma API. Existem APIs que funcionam em sistemas operacionais para facilitar as tarefas de comunicação e de processamento do mesmo, portanto não precisam da utilização da internet.

Com o uso de uma API não importa se deseja construir uma aplicação web, móvel ou *desktop*, pois ela vai servir de interface para qualquer aplicação cliente que deseje construir. Além de proporcionar integração, as APIs têm um papel fundamental na programação distribuída, pois permite que qualquer cliente, com conhecimento sobre a documentação da mesma, possa consumir, sem ter que criar novamente uma mesma lógica e simplesmente aproveite uma já existente, evitando trabalhos redundantes e podendo aproveitar de

serviços consolidados e inovadores, como as APIs do *Google Maps* [1], do *Facebook* [2], da *Amazon* [3], do *PayPal* [4] e entre outras.

Os *softwares* dos dias atuais têm uma demanda que só pode ser resolvida com conceitos e técnicas de aplicações distribuídas. Conceitos como disponibilidade e confiabilidade, somente aplicações distribuídas podem oferecer. Com o aparecimento e a consolidação, das diversas maneiras de acessar uma aplicação, é inviável a construção de *softwares* que não permita a interoperabilidade e a escalabilidade, que uma aplicação distribuída pode disponibilizar.

Com a crescente utilização da internet, os *softwares* devem lidar com um volume crescente de usuários e devem continuar a funcionar com desempenho satisfatório. Deste modo, distribuir uma aplicação torna-se uma necessidade desafiadora para arquitetos ou desenvolvedores de *software* (PORTO et al., 2009).

Além da questão da interoperabilidade e da escalabilidade, que são essenciais para ser implementados em uma aplicação distribuída, existem outras preocupações que são de fundamental importância, que é a segurança e o tempo de resposta. Como garantir a segurança dos dados de uma aplicação, que estarão funcionando de forma distribuída em um *cluster* ?. Nenhuma empresa vai confiar os seus dados em um *software*, se não tiver a convicção que eles estarão seguros. Além disso, uma aplicação distribuída pode ter partes espalhadas por servidores em lugares muito distantes. Como diminuir o tempo de resposta de uma requisição, para não atrapalhar a experiência do usuário com a aplicação ?. São dois questionamentos muito importantes, a serem feitos antes de começar a implementação de uma aplicação distribuída.

Uma das tendências que mais cresce ao criar uma aplicação nos dias atuais, é a separação de responsabilidades no processo de desenvolvimento, ou seja, o processo ocorre em duas frentes, o *front-end*, responsável pela interface e experiência do usuário, e o *back-end*, responsável pelo gerenciamento de dados e dos servidores. Essa tendência segue justamente um dos princípios estabelecidos com o REST, que prega que desde o início do desenvolvimento de uma aplicação, a divisão de responsabilidades deve estar claro, para proporcionar assim a criação de camadas descentralizadas que futuramente podem se comunicar e funcionar como uma aplicação distribuída.

Para alcançar a criação de uma aplicação distribuída, que atenda a todas propriedades que um *software* necessita e permita a utilização dessa abordagem de desenvolvimento, foi proposto a criação de uma API no estilo arquitetural REST, que implementa todos os requisitos de uma aplicação e que vai proporcionar o funcionamento dessa aplicação

[1] https://developers.google.com/maps
[2] https://developers.facebook.com
[3] https://developer.amazon.com/services-and-apis
[4] https://www.paypal-brasil.com.br/desenvolvedores

de forma distribuída. Com a utilização de um API, passamos a ter um alto poder de integração, pois é possível criar várias interfaces para vários dispositivos, seja Web, móvel ou *desktop* e em qualquer linguagem, sem a necessidade de refazer o mesmo trabalho de gestão de dados, para cada plataforma diferente. A gestão de dados irá se tornar mais simples, já que a única forma de acessar toda a infraestrutura criada será por meio da API, então pode-se criar relatórios do uso dos dados e com isso fazer estudos e traçar perfis de quem está usando os serviços oferecidos e melhorar a relação com os usuários da plataforma, e por último, podemos distribuir os serviços da aplicação para outros desenvolvedores e empresas.

O objetivo deste trabalho portanto, é a construção de uma API que se baseia no estilo arquitetural REST para a implementação dos requisitos de uma aplicação, o *Language Adviser*, e permitir que, a partir desta, proporcione a integração com qualquer aplicação cliente, desenvolvida em qualquer linguagem, e que forneça requisitos como segurança, escalabilidade, robustez e rapidez, para que a mesma consiga ter um ótimo rendimento, quando for colocada em um ambiente real. Para facilitar o entendimento da API e como ela está funcionando, foi proposto também a criação de uma documentação para que os desenvolvedores *front-end* consigam consumir a API.

Com este trabalho espera-se comprovar os benefícios de utilizar uma API, seguindo os princípios do estilo arquitetural REST, como ferramenta ideal para uniformizar comunicações e promover serviços distribuídos. Além da utilização do REST, como ferramenta para proporcionar que o *Language Adviser* se torne uma aplicação distribuída, espera-se que a aplicação possua propriedades que a torne robusta, como por exemplo, a possibilidade de escalabilidade, de segurança e gerência de dados.

Tendo como base os requisitos da aplicação *Language Adviser*, a API foi desenvolvida em *JavaScript*, executada na plataforma Node.js e com o auxílio do *framework Express* que se conecta e interage com dois bancos de dados NoSQL distintos, o *ArangoDB* e o Redis. O *ArangoDB* é o banco de dados responsável pelo armazenamento dos dados e o Redis, é para que a aplicação *Language Adviser*, possua um gerenciamento de cache, para tornar a busca pelos dados mais rápida. Para garantir a segurança dos dados, foi implementado um esquema de autenticação baseado em *token*, que proporciona para o servidor o não gerenciamento de estados de usuários e também conta com a presença de um módulo em Node.js que facilita o escalonamento dos processos da API.

O presente documento está organizado da seguinte maneira: no Capítulo 2 são apresentados os principais conceitos para a criação da API e as tecnologias que foram utilizadas; no Capítulo 3 é exposto a visão geral do *Language Adviser* e como foi feita a sua implementação em uma aplicação distribuída; no Capítulo 4 é apresentado a prova de conceito e os resultados da API; no Capítulo 5 são apontadas as considerações finais; e no Apêndice A exibe todos os recursos e rotas disponíveis na API.

# 2 Referencial Teórico

## 2.1 *Web Services*

A interoperabilidade é uma das mais importantes características ao desenvolver uma aplicação. As aplicações precisam se comunicar com outras para obter o acesso a informações que deseja e também para permitir o desenvolvimento de aplicações descentralizadas e distribuídas. A capacidade de duas aplicações se comunicarem em máquinas diferentes, sendo executadas em sistemas operacionais distintos e de atuar em conjunto, é denominada interoperabilidade. Segundo Sheth (1999), a nova geração de sistemas de informação deverá ser capaz de resolver a interoperabilidade, para poder fazer um bom uso das informações disponíveis.

No final da década de 1990, com a crescente utilização da internet e diante da necessidade que diferentes aplicações pudessem se comunicar, surgiu uma nova nomeação para sistemas distribuídos que permite a troca de dados na internet, que são os chamados *web services* (MORAES et al., 2013). O *web service* é uma nomeação para tecnologias que permite a interoperabilidade com qualquer outra aplicação através da internet. Para Sousa (2015), a necessidade de criar uma forma de comunicação entre diferentes aplicações, de modo que aplicações hospedadas em diferentes máquinas e em redes distintas fossem capazes de se comunicar, levou a criação de tecnologias *web services*.

Os *web services* são aplicações que se comunicam pela internet, implementadas de forma flexível e fracamente acoplada, que estão tendo um papel crescente nas interações entre empresas no meio eletrônico (BAX; LEAL, 2001).

De acordo com a *World Wide Web Consortium* (W3C), que é um consórcio de empresas que tem o objetivo de criar padrões para a internet, um *web service* define-se como um sistema de software projetado para suportar a interoperabilidade entre máquinas na rede (W3C, 2018).

O *web service* nada mais é que, um sistema em camadas que funciona no lado do servidor e fornece uma ligação padrão para conectar as funcionalidades do mesmo, com distintos *softwares* (GURUGÉ, 2004). Os *web services* funcionam com a utilização do protocolo da camada de aplicação HTTP. Os clientes que desejam consumir os *web services*, só precisam fazer requisições para o servidor que está hospedando o mesmo.

Segundo Gurugé (2004), um *web service* não é feito para ser uma aplicação completa, e sim tornar a programação distribuída e acessível para qualquer programa que necessite dos serviços que o mesmo oferece.

Como a internet estava se tornado um meio de comunicação cada vez mais presente na vida das pessoas surgiu a necessidade de padronizar comunicações entre diferentes plataformas (Mac, *Windows*, Linux) e entre diferentes linguagens de programação (MORAES et al., 2013). Essa foi uma das razões que proporcionou a criação de diversos padrões e estilos arquiteturais de *web services*.

Atualmente os mais populares e que estão disponíveis para a comunidade são o *GraphQL*, que é um estilo arquitetural desenvolvido pelo Facebook, onde representa todos os dados como se fosse um gráfico e possui uma própria linguagem de consulta; o *Falcor* desenvolvido pela Netflix é uma biblioteca *JavaScript* que tem a premissa de entregar os dados somente que o cliente expressou na consulta e também a criação de um modelo virtual de armazenamento de dados; o *web service* SOAP foi um padrão que ficou bastante conhecido e foi utilizado por várias empresas, utilizava-se do *eXtensible Markup Language* (XML) e do envio de mensagens SOAP, na maior parte dos casos era manuseando em conjunto com o protocolo HTTP. Atualmente o estilo arquitetural mais popular é o REST, essa popularidade sé da pela sua semelhança de funcionamento com a internet e por se utilizar de conceitos já estabelecidos pelo protocolo HTTP.

Com a popularização da internet e a crescente utilização das APIs pelas empresas, permitiu que as tecnologias *web services* se desenvolvesse e tornasse um item obrigatório para qualquer aplicação que deseja alcançar escalabilidade e a distribuição dos seus serviços.

### 2.1.1 Protocolo HTTP

O *Hypertext Transfer Protocol* (HTTP) é a base sobre qual a internet funciona, e os *web services* utilizam-se do próprio, como o mecanismo de transporte para o envio e o recebimento de mensagens. O protocolo HTTP é baseado em requisições e respostas entre cliente e servidor.

Para fazer uma requisição para o servidor, é preciso, entre outras coisas, que na mensagem esteja especificado qual método será utilizado para o envio da requisição. Os métodos HTTP indicam qual a ação a ser realizada no servidor. O protocolo HTTP define nove métodos, mas apenas quatro são mais utilizados, os quais são apresentados a seguir.

O método *GET* requisita um determinado recurso e deve ser usado apenas para recuperar dados e não para modificar, essa sua característica de apenas recuperar dados o torna um método seguro. O método *POST* envia dados a serem processados no servidor, é utilizado para a criação de novas informações. O método *PUT* atualiza informações existentes. O método *DELETE* é o responsável pela exclusão de informações.

Toda resposta do protocolo HTTP possui um código de estado. Esse código indica ao cliente sobre o que ocorreu com a requisição no servidor. Com os códigos de estado do HTTP é possível saber se uma operação foi realizada com sucesso, se uma informação não

existe mais, se ocorreu algum conflito, se teve algum problema interno ao servidor, se não possui a autorização para acessar e entre outros. Na Tabela 1 exibimos a categoria dos códigos de estado do HTTP.

Tabela 1 – Categorias dos códigos de estado do HTTP

| CATEGORIA | DESCRIÇÃO |
|---|---|
| 1xx:Informativa | Comunica informações do protocolo de transferência. |
| 2xx:Sucesso | Indica que a requisição do cliente foi aceita com êxito. |
| 3xx:Redirecionamento | Indica que o cliente deve tomar alguma ação adicional para completar seu pedido. |
| 4xx:Erro de Cliente | Indica os casos em que o cliente pode ter cometido algum erro. |
| 5xx:Erro de Servidor | Indica um erro do servidor ao processar a requisição. |

Fonte – (MASSE, 2011)

### 2.1.2 REST

*Representational State Transfer* (Representação por Transferência de Estado) foi apresentada inicialmente na tese de doutorado de Roy Thomas Fielding no ano de 2000. O REST é um estilo arquitetural para sistemas distribuídos que especifica regras e introduzem propriedades como escalabilidade, cache e identificação de recursos, permitindo o desenvolvimento de uma aplicação distribuída no seu estado da arte.

Segundo Fielding (2000) o REST é um estilo arquitetural para sistemas distribuídos e hipermídias. A composição desse novo estilo arquitetural parte de um estilo que Fielding nomeia de "Desconhecido" e adiciona características, que são as chamadas restrições REST, que também foi estudado por ele em sua tese de doutorado (RIBEIRO; FRANCISCO, 2016).

> Examinando o impacto de cada restrição, como isso é adicionado ao estilo referenciado, pode-se identificar as propriedades induzidas pelas limitações da Web. Restrições adicionais podem então ser aplicadas para formar um novo estilo arquitetural que melhor reflita as propriedades desejadas de uma arquitetura Web (FIELDING, 2000)

As restrições do REST têm como objetivo determinar a forma na qual padrões como o HTTP e o URI devem ser modelados. Conforme afirma Cavaleiro (2013), diferente dos serviços baseados em SOAP, o REST utiliza o protocolo HTTP como parte principal e fundamental da sua arquitetura. Ao todo são seis restrições que foram pensadas para fazer parte do estilo arquitetural. As restrições (*constraints*) do REST são:

1 – Cliente e Servidor: É a restrição básica para uma aplicação REST. A arquitetura cliente e servidor é bastante conhecida e utilizada em aplicações web e no REST ela tem o objetivo de separar as responsabilidades. O cliente que vai consumir o serviço não precisa se preocupar com tarefas como banco de dados, cache, escalabilidade, entre outros. Isso

permite a evolução independente de cada um dos módulos facilitando a manutenção (COSTA et al., 2014).

Para Ribeiro e Francisco (2016), a atenção focada na portabilidade de interfaces em diferentes plataformas, alicerçada na arquitetura cliente e servidor, é fundamentada no conceito de *web services*, que oferecem o "*write once, run anywhere*" (escreva uma vez, execute em qualquer lugar). Como a camada de negócio já está desenvolvida no servidor e disponibilizada via HTTP, qualquer cliente seja web, *desktop* ou móvel, pode se comunicar com essa camada (RIBEIRO; FRANCISCO, 2016). De acordo com Fielding (2000):

> Ao separar as preocupações de interface de usuário das preocupações de armazenamento de dados, podemos melhorar a portabilidade da interface do usuário em diversas plataformas e elevar a escalabilidade por meio da simplificação dos componentes do servidor.

2 – *Stateless*: Essa restrição propõe que cada requisição ao servidor não deve ter qualquer ligação com nenhuma outra requisição. *Stateless* significa um protocolo sem estado, ou seja, todas as requisições são independentes e deve conter o mínimo de informação necessária para que o servidor consiga compreender o pedido feito pelo cliente. Aplicações REST não podem guardar informações relativas aos usuários por meio de sessões e *cookies*. *Stateless* significa que toda solicitação HTTP ocorre em um isolamento completo e o servidor nunca pode armazenar as informações das solicitações anteriores (RICHARDSON; RUBY, 2008). Segundo Ngolo (2009) a independência entre as requisições, além de prover a escalabilidade vai permitir que recursos computacionais possam ser distribuídos em diferentes servidores, ideal para um cenário de balanceamento de carga.

3 – Cache: Quando vários clientes acessam um mesmo servidor em busca de dados é comum utilizar-se de cache para acelerar as respostas e evitar processamento desnecessário. Em uma API REST, para obter uma melhor performance, é ideal a utilização de cache. Uma aplicação que se utiliza de cache vai evitar comunicações desnecessárias, reduzir a latência da rede e aumentar a eficiência, a escalabilidade e a percepção da performance (FIELDING, 2000).

4 – Interface Uniforme: É a principal restrição do REST. A utilização de uma interface uniforme é o que o distingue de todos os outros *web services*. O uso de interface uniforme tem o foco na simplicidade, acessibilidade, interoperabilidade e na capacidade de descoberta de recursos (COSTA et al., 2014). O importante sobre a interface uniforme é garantir que todo serviço use a interface do HTTP do mesmo modo, para que possamos ter serviços mais facilmente compreensíveis (RICHARDSON; RUBY, 2008). Com o objetivo de obter uma interface uniforme foi definido quatro requisitos, que são:

1. Identificação de Recursos: Recursos são elementos de informação, que através de um identificador podem ser manipuladas. A identificação ocorre por meio do *Uniform Resource Identifiers* (URI). Segundo Coulouris, Dollimore e Kindberg (2012) um URI é um conjunto de caracteres que servem para identificar um recurso na internet. Qualquer componente da aplicação que necessita de ter informações é considerado um recurso, um recurso precisa ter uma URI para ser endereçada (SOUSA, 2015).

2. Representação de Recursos: A forma como os recursos são representados para os clientes é bastante importante, para que o cliente consiga trabalhar com esses recursos de forma correta. A representação dos recursos pode ser em *JavaScript Object Notation* (JSON), *eXtensible Markup Language* (XML), *Hypertext Markup Language* (HTML) e entre outras. O que importa nesse requisito é que aplicação tem que deixar claro como o recurso vai ser representado.

3. Mensagens Autoexplicativas: Para que o servidor ou o cliente consiga entender o que foi enviado na requisição ou na resposta, é necessária a passagem de metadados. Entre os metadados mais importante para o HTTP estão o código e o método HTTP, *host, media type*, idioma e a compressão utilizada. As APIs REST trabalham em cima do HTTP, então o cabeçalho do HTTP é fundamental tanto para o servidor como para o cliente.

4. *Hypermedia As The Engine Of Application State* (HATEOAS): Uma determinada API REST deve ser navegável, a partir de um recurso, deve descobrir quais recursos lhe estão associados (SOUSA, 2015). Conforme afirma Cavaleiro (2013) o princípio do HATEOAS define como uma aplicação cliente pode interagir com uma aplicação em rede através de hipermídias. As respostas das APIs REST devem conter além dos dados solicitados em uma requisição, *hyperlinks* contendo informações sobre os recursos disponíveis da API. O uso do HATEOAS no funcionamento do REST, o difere de todos os outros *web services* (BURKE; REDDY; SRINIVASAN, 2010). Na Figura 1 temos um exemplo da representação do HATEOAS em uma resposta HTTP no formato JSON.

Figura 1 – Representação do HATEOAS

```
{
  "_links": [
    {
      "rel": "Add User",
      "method": "POST",
      "href": "/user"
    },
    {
      "rel": "Edit User",
      "method": "PUT",
      "href": "/user/10"
    },
    {
      "rel": "Delete User",
      "method": "DELETE",
      "href": "/user/10"
    }
  ]
}
```

Fonte: Autoria Própria

5 – Sistema em Camadas: A utilização de camadas para separar diferentes unidades de funcionalidade de uma aplicação é extremamente importante. Com o intuito de difundir a criação de aplicações distribuídas, uma API REST deve ser composta em camadas. Com a criação de camadas ganha-se em desempenho, confiabilidade e processamento (MORO; DORNELES; REBONATTO, 2009).

6 – Código Sob Demanda: É a única restrição opcional proposta por Fielding. Essa restrição permite que o cliente possa executar algum código sob demanda, dessa forma diferentes clientes se comportam de maneiras distintas, mesmo utilizando exatamente os mesmos serviços oferecidos pelo servidor (FIELDING, 2000). A ideia era de aumentar a flexibilidade do lado do cliente.

### 2.1.3 Modelo de Maturidade de Richardson

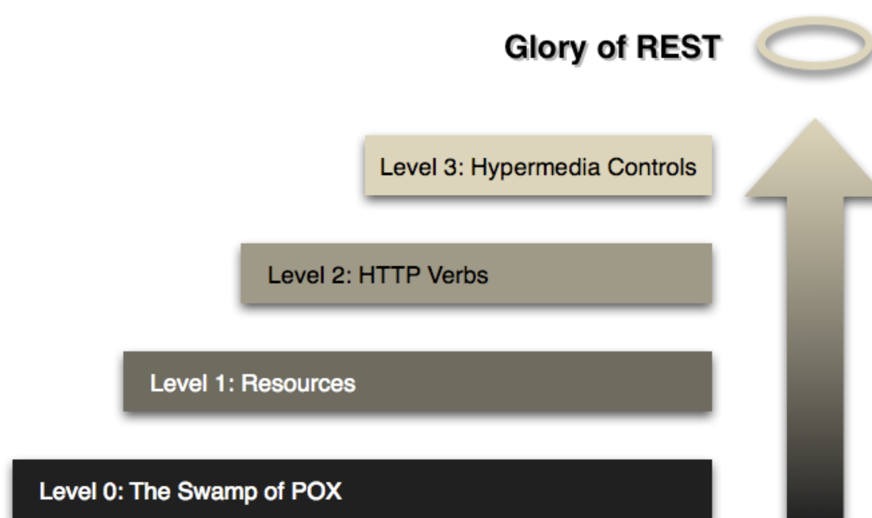
Quando estamos falando do estilo arquitetural devemos utilizar o termo REST, já no momento da implementação do estilo arquitetural temos que utilizar a denominação RESTful (FOWLER, 2010). Para Fielding (2008) uma API só pode ser considerada RESTful, se seguir todas as restrições definidas no estilo arquitetural REST.

De fato, o principal objetivo dos *web services* RESTful é assegurar que as aplicações sejam mais próximas e parecidas com a web. Dessa forma, os sistemas computacionais podem ser mais escaláveis, de fácil modificação e com performance melhorada para

transações (RICHARDSON, 2008).

Com o objetivo de tornar a implementação mais simples e de difundir o estilo arquitetural REST, Richardson (2008) propôs um modelo em quatro níveis para alcançar a "glória" do REST. O modelo proposto por Richardson (2008) ficou conhecido como o modelo de maturidade de Richardson. Cada nível proposto exige a implementação das restrições REST, sendo que o atendimento das implementações requeridas nos níveis anteriores são pré-requisitos ao nível seguinte (RIBEIRO; FRANCISCO, 2016). Na Figura 2 temos a classificação dos quatro níveis propostos por Richardson (2008) para alcançar a "glória" do REST.

Figura 2 – Níveis para a glória do REST




Fonte: (FOWLER, 2010)

O nível 0, pântano do POX, é a ausência de todas as restrições definidas no REST, em que só se utiliza do HTTP como mecanismo de transporte. O POX é um estilo de programação sem nenhuma preocupação com a definição de recursos e do uso correto dos métodos e dos códigos de estado do HTTP. Na Tabela 2 temos uma demonstração de como é realizado a modelagem de um API RPC POX.

Tabela 2 – Modelo RPC POX

| MÉTODO | URI | OBJETIVO |
|---|---|---|
| POST | /getUsers | Ler uma lista de usuários |
| POST | /saveUsers | Salvar um usuário |
| POST | /editUsers | Editar um usuário de id 10 |
| POST | /deleteUsers | Excluir um usuário de id 10 |

Fonte – Autoria Própria

O nível 1 é a utilização de recursos. A API será modelada com base nos recursos disponíveis, onde os recursos em REST são os componentes da API que precisam estar disponíveis para comunicação (FOWLER, 2010). Conforme afirma Richardson e Ruby (2008), cada coisa interessante gerenciada pela aplicação deve ser exposta como um recurso. Segundo Fowler (2010) a adição do nível 1 faz com que em vez de todos os pedidos ir a um simples *endpoint*, o que ocorre nos *web services* SOAP, a interação ocorra com os recursos individuais disponíveis por meio da URI de cada recurso.

O nível 2, verbos do HTTP, exige a utilização dos verbos ou métodos do HTTP para diferentes tipos de operações nos recursos disponíveis da API. Além do uso correto dos métodos HTTP, os códigos de estado também devem ser usados corretamente, para indicar ao cliente o estado da sua requisição. Supondo uma aplicação que tenha disponível um recurso chamado *users*, utilizando os métodos HTTP, podemos acessar esse recurso de diferentes maneiras. A Tabela 3 mostra as operações que ocorre em uma API REST de acordo com o método utilizado na requisição.

Tabela 3 – Modelo REST

| MÉTODO | URI | OBJETIVO |
|---|---|---|
| GET | /users | Ler uma lista de usuários |
| POST | /users | Salvar um usuário |
| PUT | /users/10 | Editar um usuário de id 10 |
| DELETE | /users/10 | Excluir um usuário de id 10 |

Fonte – Autoria Própria

O nível 3, controle de hipermídias, é o uso do HATEOAS na API, para permitir que o cliente consiga ter uma maior interação com a aplicação, sem precisar conhecer a documentação. Para Fielding (2000) a adoção do HATEOAS é fundamental para as aplicações REST. Se o motor do estado do aplicativo (e, portanto, a API) não é conduzido por hipertexto, então não pode ser RESTful e não pode ser uma API REST (FIELDING, 2008).

Um *web service* que se utilize do HTTP como protocolo, URI únicas para cada recurso, alternância entre serviços dos recursos pela utilização dos métodos do HTTP e a utilização de hipermídias implementaria a "glória" do REST (RICHARDSON, 2008).

## 2.2 NoSQL

Banco de dados não relacionais (NoSQL) surgiram como alternativa à questão de escalabilidade no armazenamento e no processamento de grandes volumes de dados

(DIANA; GEROSA, 2010). Conforme afirma Sousa (2016), o movimento NoSQL é recente e vem crescendo rapidamente e dentre os fatores está a quantidade de dados gerados pelas aplicações web, que não estava mais tendo um bom funcionamento no modelo relacional. As aplicações de hoje em dia visam garantir a eficiência, no modelo relacional: quanto maior a quantidade de dados armazenados, maior o tempo de resposta, o que não agradava muito os desenvolvedores que trabalhava com uma quantidade de dados considerável. Conforme afirma Diana e Gerosa (2010) antes do surgimento do movimento NoSQL, grandes empresas enfrentavam o problema de gerenciar grande quantidade de dados, esses problemas eram resolvidos pelas próprias empresas que criavam suas próprias soluções, o que acabou contribuindo para o surgimento dessa nova categoria de banco de dados.

Os bancos de dados NoSQL apresenta algumas características em comum, as principais são a utilização do BASE ao invés do ACID, possuir grande escalabilidade horizontal, baixa latência, maior disponibilidade de acesso e possuir linguagens de consultas próprias que são adequadas ao seu modelo de dados.

Apesar das inúmeras características em comum, cada banco de dados NoSQL possui seu próprio modelo de dados. Os principais modelos de dados são o chave e valor, orientado a documentos, orientado a colunas e orientado a grafos (HOLANDA; SOUZA, 2015). Para Sousa (2016) não existe um modelo superior ao outro, tudo depende do contexto em que vai ser utilizado.

Bancos de dados chave e valor é todo composto por um conjunto de chaves, as quais estão associadas a determinados valores (LÓSCIO; OLIVEIRA; PONTES, 2011). Os bancos de dados chave e valor são ideais para aplicações com um alto consumo computacional.

O modelo de dados orientado a documentos, permite o armazenamento de coleções de documentos. Uma coleção no modelo não-relacional se refere a uma tabela no modelo relacional, só que uma coleção não possui um esquema relacionado, tendo assim uma maior flexibilidade. Um documento é um objeto com um identificador único e um conjunto de campos (LÓSCIO; OLIVEIRA; PONTES, 2011). Os documentos armazenados geralmente estão no formato *JavaScript Object Notation* (JSON).

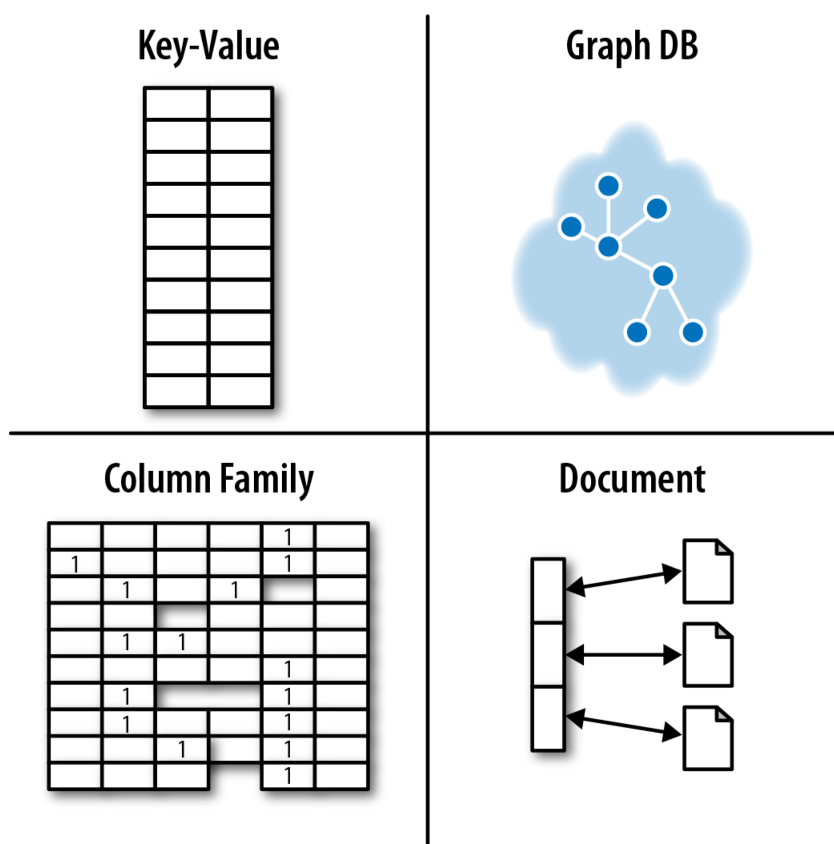
Para Abadi, Boncz e Harizopoulos (2009), o modelo colunar de armazenamento mantém cada coluna do banco de dados armazenada separadamente, guardando contiguamente os valores de atributos pertencendo à mesma coluna de forma densa e comprimida. As principais vantagens que se pode obter com a arquitetura de armazenamento dos bancos de dados do modelo colunar está vinculada à compressão, materialização e blocos de iteração (SOARES; BOSCARIOLI, 2012).

Os bancos de dados orientados a grafos se caracterizam por possuírem estruturas de dados no qual os esquemas são modelados como grafos (ANGLES; GUTIERREZ, 2008). O grafo consiste de vértices e arestas, em que os vértices atuam como entidades e as arestas

como os relacionamentos entre as entidades (BARRETO, 2017). No modelo orientado a grafos, como os dados são organizados em relacionamentos, facilita a execução de consultas complexas.

A combinação de mais de um modelo de dados resulta em um banco de dados hibrido. Um banco de dados hibrido utiliza-se de vários modelos em conjunto para realizar o armazenamento de seus dados e relacionar a outros modelos dentro do mesmo banco de dados (SOUSA, 2016). Na Figura 3, temos a ilustração de todos os modelos de dados do NoSQL.

Figura 3 – Modelos de Dados




Fonte: (ROBINSON; WEBBER; EIFREM, 2015)

### 2.2.1 *ArangoDB*

O *ArangoDB* é um banco de dados NoSQL desenvolvido pela ArangoDB GmbH e lançado em 2011. É um banco de dados híbrido, pois combina vários modelos entre eles o orientado a grafos, orientado a documentos e o chave e valor. Seu modelo de persistência é determinado automaticamente, dessa forma o usuário pode modelar seus dados tanto como documento, pares chave e valor e orientado a grafos (SOUSA, 2016).

Os dados do *ArangoDB* são representados na forma de JSON e transportado via HTTP. Os documentos do ArangoDB podem conter um ou mais atributos de diferentes valores, os principais valores disponíveis no ArangoDB são: número, *string, booleano, null, array* e objeto.

O *ArangoDB* é um banco de dados com diversas características que o diferencia dos demais bancos de dados NoSQL. Dentre as principais características do *ArangoDB*, destacam-se:

1. Modelo de dados flexível.
2. Linguagem de consulta própria, *ArangoDB Query Language* (AQL).
3. Transações, permite usar o ACID.
4. Replicação.
5. Código aberto.

O tipo de consistência utilizada nos bancos de dados NoSQL é chamada de consistência eventual ou BASE. O *Basically Available, Soft state, Eventual consistency* (BASE) se caracteriza por estar basicamente disponível e eventualmente consistente, ao contrário dos bancos de dados relacionais, que se utilizam do *Atomicity, Consistency, Isolation, Durability* (ACID) para garantir que o banco de dados estejam sempre consistente (BRITO, 2010). Apesar de ser um banco de dados NoSQL, o ArangoDB permite utilizar transações ACID, caso seja necessário, além da escalabilidade horizontal e vertical, dando uma vasta gama de opções para os desenvolvedores (ARANGODB, 2018).

Além de permitir a utilização do ACID, o que é uma vantagem em relação aos demais bancos de dados NoSQL, outro ponto forte do ArangoDB é o fato de possuir uma linguagem de consulta própria ao seu modelo de dados. A *ArangoDB Query Langauge* (AQL) é bastante similar a *Structured Query Language* (SQL) e para Sousa (2016) ela tem o mesmo objetivo da SQL de ser bastante legível e de fácil aprendizagem para humanos.

Para a realização de uma consulta na AQL, utiliza-se basicamente de duas expressões: *FOR* e o *RETURN*. A expressão *FOR*, é o responsável por percorrer todos os documentos cadastrados em uma coleção, o *FOR* tem a mesma função que o *SELECT* para o SQL. A expressão *RETURN* é usado para determinar quais atributos ou se todo o documento será retornado de uma consulta. Caso deseje ter uma consulta mais refinada existem outras expressões como o *FILTER*, *ORDER BY* e entre outros.

Na Figura 4 temos a demonstração de uma consulta AQL, nessa consulta estamos acessando a coleção *users* e filtrando o resultado para mostrar somente usuários com um salário maior e igual do que 1000.

Figura 4 – Consulta AQL

```
FOR user IN users
FILTER user.salary >= 1000
RETURN user
```

Fonte: (ARANGODB, 2018)

Além do uso da AQL, o ArangoDB permite o uso de funções em *JavaScript* para fazer buscas mais aprofundadas e completas em suas coleções. O AQL vem com um conjunto integrado de funções, mas não é uma linguagem de programação completa. Para adicionar funcionalidade ausente ou simplificar consultas, os usuários podem adicionar funções escritas em *JavaScript* (ARANGODB, 2018).

O *ArangoDB* possui várias características o que torna um banco de dados muito atrativo para ser utilizado. Além das inúmeras funcionalidades que o *ArangoDB* dispõe, ele também é muito bom em termos técnicos em comparação a outros bancos de dados NoSQL. Nos testes realizado por Oliveira e Cura (2017) o *ArangoDB* foi o melhor banco de dados NoSQL, nos quesitos de inserção de documentos e de recuperação de documentos, à frente de banco de dados como o *MongoDB* e o *OrientDB*.

### 2.2.2 Redis

O Redis é um banco de dados chave e valor de código aberto e é o mais popular do mundo nessa categoria. O Redis possui uma estrutura de chave e valor que consegue salvar vários tipos de dados, como *string, hash, list, set, sorted set*. Conforme afirma Carlson (2013) suas estruturas de dados podem resolver grande parte dos problemas de armazenamento sem a necessidade de mais implementações com outros bancos de dados.

Segundo Sampaio e Knop (2015) os bancos de dados chave e valor podem ser divididos em duas categorias, os que armazena em memória ou em disco. O armazenamento em memória possui alto desempenho, normalmente usados para cache, e o armazenamento em disco, são usados como banco de dados padrão (CARLSON, 2013).

O Redis é um banco de dados chave e valor hibrido, pois trabalha armazenando os dados em memória e em disco. Para alcançar um excelente desempenho, o Redis trabalha em maior parte do seu processamento com um conjunto de dados na memória do dispositivo que o está executando, em contraste da maioria dos bancos de dados, que processa maior parte do tempo no disco. Dependendo da quantidade de dados em que esteja trabalhando, o Redis oferece a opção de desejar um conjunto de dados no disco em um período de tempo, para não sobrecarregar a memória. O Redis é implementado na linguagem C e é

*single-thread*, e possui um servidor em rede que recebe requisições de clientes através de uma porta TCP/IP (BOCHI, 2015).

O principal caso de uso do Redis é o armazenamento de cache por causa do seu desempenho. O Redis inserido na "frente" de outro banco de dados, cria um cache na memória com excelente desempenho, diminuindo a latência de acesso e aumentando o *troughput* (BOCHI, 2015).

O Redis é uma ferramenta que está ganhando muita popularidade nos últimos anos, graças as características que ele dispõe e também ao seu ótimo desempenho. Nos últimos anos, empresas como Instagram e Github passaram a utilizar o Redis, o qualificando como ótima escolha para o ter como o banco de dados ideal para o gerenciamento de cache.

## 2.3 Node.JS

O Node.js é uma plataforma para aplicações web desenvolvida por Ryan Dahl no ano de 2009 e construída sobre o motor *JavaScript* do navegador *Google Chrome*. O aparecimento de projetos como Node.js permitiram tornar o *JavaScript* uma linguagem possível de ser usado na implementação de componentes de um servidor de uma aplicação Web (SEVERANCE, 2012).

O *JavaScript* é uma linguagem de programação interpretada, multi-plataforma e orientada a eventos sendo desenvolvido em 1995 por Bredan Eich e foi utilizado pela primeira vez nos *browsers Netscape* (SEVERANCE, 2012). Segundo Sousa (2015) o *JavaScript* foi desenvolvido com foco em disponibilizar mais funcionalidades das páginas web, tornando as mesmas, mais interativas. O *JavaScript* até o surgimento do Node.js era sempre executado no lado do cliente por meios dos *browsers*.

O Node.js usa um modelo de I/O direcionada a evento não bloqueante o que torna leve e eficiente. O principal objetivo do Node.js era resolver os problemas das arquiteturas bloqueantes e com isso aumentar a escalabilidade das aplicações. Em linguagens como Java e PHP cada conexão com o servidor cria uma nova *thread*, com isso quanto maior o número de usuários conectados, maior o custo de processamento associado ao servidor (SOUSA, 2015).

O Node.js resolve esse problema, tratando diferente a forma de criar conexões, ao invés de para cada conexão criar uma nova *thread*, ele dispara um evento dentro do motor de processos, permitindo assim suportar uma grande quantidade de usuários conectados, sem a necessidade de fazer um escalonamento horizontal e dificultar todo o gerenciamento da aplicação.

O motor do navegador *Google Chrome*, o V8 *JavaScript Engine*, é um interpretador implementado pelo Google em C++ e é a base sobre o qual o Node.js foi construído. A

proposta do V8 é acelerar o desempenho de uma aplicação compilando código *JavaScript* diretamente para código binário antes de executá-lo, permitindo que um código *JavaScript* seja tão rápido, quanto um código escrito em uma linguagem de baixo nível (SEVERANCE, 2012). A V8 é extremamente rápida e tem um bom desempenho em diversas circunstâncias para aplicações em *JavaScript* (HUGHES-CROUCHER; WILSON, 2012).

O desenvolvimento de aplicações com o Node.js é feito utilizando uma abordagem orientada a eventos. A programação orientada a eventos se difere dos demais paradigmas de programação que seguem um fluxo padronizado. Na abordagem orientada a eventos o fluxo do código é indicado por meio de disparos, ou indicações externas, chamado de eventos (SEVERANCE, 2012). O desenvolvedor precisa conhecer os eventos que serão disparados, para que consiga saber como tratar esse evento e realize as operações necessárias.

Com a utilização da abordagem orientada a eventos, ocorre o uso da arquitetura *Event Loop*. Segundo BARRETO (2017) o *Event Loop* é o mecanismo interno responsável por escutar e emitir eventos no sistema. O uso do *Event Loop* faz com que o Node.js se utilize de operações I/O não bloqueantes e assíncronas, apenas registrando a funções de *callback* para indicar o estado da respectiva função (SOUSA, 2015).

Segundo Abernethy (2011) o Node.js é extremamente bem projetado para situações em que um grande volume de tráfego é esperado e o processamento necessário do lado do servidor é volumoso. O Node.js é uma poderosa plataforma, sendo muito eficiente e indicado para alguns casos, como em *streaming, Internet of Things* e principalmente para aplicações em tempo real e para o desenvolvimento de APIs, onde ocorre uma grande troca de dados, sendo o cenário ideal para ter uma plataforma leve e rápida.

Conforme afirma BARRETO (2017) o Node.js é composto por módulos que fazem parte do núcleo e módulos criados pela comunidade. Por meio do *Node Package Manager* (NPM), é possível fazer a integração com vários módulos disponibilizados pela comunidade.

### 2.3.1 *Express*

O módulo mais conhecido para configurar o ambiente de desenvolvimento e auxiliar as tarefas de implementação do serviço web do Node.js é o *framework Express*. O *Express* é um *framework* desenvolvido em *JavaScript* e disponibilizado via NPM. De acordo com Hahn (2016) o *Express* abstrai a complexidade do Node.js adicionando um número significativo de funcionalidades, entre elas, gerenciar pedidos com métodos HTTP em diferentes rotas, interagir com mecanismos de renderização, configuração de diferentes módulos e entre outras. O desenvolvimento de APIs com o *Express* permite a criação de servidores que receba requisições HTTP de forma simples e rápida (BARRETO, 2017).

## 2.4 JWT

Com o crescimento de equipamentos conectados, que se comunicam a sistemas, faz-se necessário que as informações que trafegam na rede sejam transmitidas de maneira segura e rápida (MONTANHEIRO; CARVALHO; RODRIGUES, 2016). Autenticação é parte importante de qualquer aplicação nos dias atuais, pois a missão de identificar quem está usando a aplicação e sem tem a permissão para usá-la, é parte fundamental para a segurança e o sucesso da mesma.

Segundo Conceicao (2015), *web services* que se utiliza do protocolo HTTP, que é um protocolo *stateless*, ou seja, não guarda o estado do usuário, é interessante usar abordagens de autenticação que seguem o principio do protocolo HTTP, para não ter futuros problemas de escalabilidade e de distribuição.

Existe várias formas para autenticar um usuário de forma *stateless*, existe o esquema de autenticação HTTP *basic* e *digest* e entre outros, porém muitas empresas atualmente querem uma forma de identificar seus usuários de uma maneira mais sútil e com a maior segurança possível. Uma das abordagens bastante utilizada atualmente é a autenticação baseada em *token*.

A autenticação baseada em *token* consiste em à API com base nas informações de um usuário gerar um *token*, que vai estar todo criptografado e enviar para o usuário. O usuário deve guardar esse *token* e toda vez que for fazer uma requisição para a API o *token* tem que ser enviado no cabeçalho da requisição, para que a API consiga autenticar o usuário e verificar se ele tem permissão para acessar aqueles dados que foi solicitado.

O *JSON Web Token* (JWT) é um método de autenticação baseado em token e é um padrão aberto com base na RFC 7519, que tem o objetivo de definir um modo compacto e independente para a transmissão de informações, através de um objeto JSON (MONTANHEIRO; CARVALHO; RODRIGUES, 2016). Um token gerado pelo JWT é constituído de três partes, separada por meio de um ponto final (.). Na Figura 5 temos o exemplo de um token gerado pelo JWT.

Figura 5 – *Token* JWT

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4
gRG9lIiwiaXNTb2NpYWwiOnRydWV9.
4pcPyMD09olPSyXnrXCjTwXyr4BsezdI1AVTmud2fU4

Fonte: (JWT, 2018)

A primeira parte do *token* JWT é conhecida como o *header*, é onde estão as

informações sobre qual algoritmo será utilizado para criptografar o *token*. O algoritmo de codificação padrão do JWT é o HS256, que é um algoritmo simétrico, que possui uma chave que é compartilhada entre as duas partes. As informações do *header* serão convertidas para gerar um *hash* único.

A segunda parte é o *payload*, que contém as informações do usuário, que irão servir depois, para a identificação. No *payload* podemos informar qualquer coisa no JSON e com base nas informações contidas no documento é que a API vai poder autenticar o usuário. As informações fornecidas para o campo *payload* serão convertidas para também gerar um *hash* único.

A terceira e última parte é o *signature* que é o responsável por indicar uma chave, que é um segredo criptografado com base no algoritmo escolhido no *header*. O *signature* é gerado a partir do *header, payload* e da chave definida pelo desenvolvedor da API (MONTANHEIRO; CARVALHO; RODRIGUES, 2016). Só quem possuir a chave definida pela aplicação vai poder gerar e criar outros *tokens* válidos.

A vantagem de utilizar o JWT em vez de abordagens como *cookies* e de sessões no servidor, é que para utilização das mesmas é necessário manter e gerir as informações de cada usuário no servidor, o que não ocorre com o JWT que apenas verifica a consistência da assinatura (CONCEICAO, 2015).

## 2.5 *OpenAPI*

A especificação *OpenAPI* define uma interface padrão independentemente da linguagem de programção para APIs RESTful, o que permite que humanos e computadores descubram e compreendam os recursos do serviço sem acesso ao código-fonte ou à inspeção de tráfego da rede. Quando adequadamente definido, um consumidor pode entender e interagir com o serviço remoto com uma quantidade mínima de lógica de implementação (OPENAPI, 2018).

O desenvolvimento eficaz de aplicações clientes de APIs REST exige que as suas interfaces estejam bem documentadas (FERREIRA et al., 2017). A *OpenAPI* possui toda uma especificação capaz de gerar uma boa documentação de uma API REST.

Na especificação da *OpenAPI* podemos criar um documento válido em dois formatos o YAML e o JSON. Dentro do esquema da especificação da *OpenAPI*, somos capazes de definir todas as rotas que existem na API, quais os métodos disponíveis em cada recurso, quais os códigos de estado disponíveis em uma resposta, além de ver o que é preciso para fazer uma requisição correta para cada rota da API. Na Figura 6 temos um exemplo da estrutura para criar uma rota usando a especificação *OpenAPI*, nesse exemplo a rota */pets* foi definido que a mesma vai atender no método HTTP GET, possuindo uma descrição

geral e as possíveis respostas que pode ser retornado para o cliente.

Figura 6 – Criação de uma rota na especificação *OpenAPI*

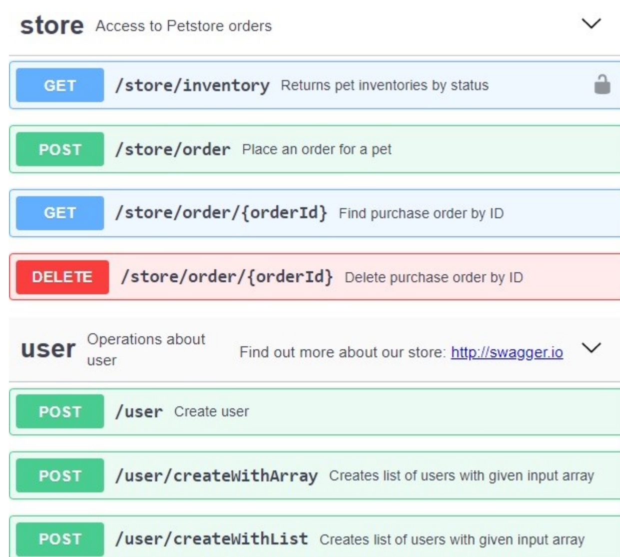
```
{
  "/pets": {
    "get": {
      "description": "Returns all pets from the system that
        the user has access to",
      "responses": {
        "200": {
          "description": "A list of pets.",
          "content": {
            "application/json": {
              "schema": {
                "type": "array",
                "items": {
                  "$ref": "#/components/schemas/pet"
                }
              }
            }
          }
        }
      }
    }
  }
}
```

Fonte: (OPENAPI, 2018)

Conforme afirma Galvao, Soares e Kai (2017) as vantagens ao uso dessa ferramenta são inúmeras, dentre as quais merecem destaque as seguintes: A documentação da API pode atrair um maior número de interessados para utilizá-la, como consequência, haverá maior facilidade em estudar a API proposta, e, por fim, será mais fácil para o desenvolvedor criar exemplos e testar a API.

Além da especificação para APIs RESTful, a *OpenAPI* possui uma ferramenta chamada *Swagger UI*, capaz de ler todo o documento escrito na especificação *OpenAPI* e gerar uma interface web. Na Figura 7 temos um exemplo de uma interface web gerada pelo *Swagger UI* contendo toda a documentação de uma API RESTful.

Figura 7 – *Swagger UI*




Fonte: (OPENAPI, 2018)

# 3 *Language Adviser*

Neste capitulo será abordado a visão do geral da aplicação, o *Language Adviser*, os seus princípios e como foi feita a implementação dos seus fundamentos em uma API RESTful, para tornar a mesma uma aplicação distribuída.

## 3.1 Visão Geral

O *Language Adviser* é uma plataforma que tem o objetivo de estabelecer o *marketing* digital através do ensino de idiomas, promovendo publicidade para seus associados. O *software* foi desenvolvido de forma que atenda todos os requisitos para facilitar o ensino e aprendizado de um novo idioma, sendo também uma grande ferramenta no processo de comunicação e persuasão de consumidores.

O *software* é uma plataforma, onde instituições de ensino de idiomas dispõem de conteúdo próprio para alimentar a plataforma, com a presença de recursos para o ensino e aprendizado dos seus usuários. A inserção de conteúdo que é realizada pelo *software*, permite que no futuro essas instituições possam alterar ou adicionar novos conteúdos para os seus alunos, sem tem que gastar esforços e recursos com a impressão de novos materiais.

Segundo Santos (2018) as instituições podem se utilizar de uma funcionalidade presente na plataforma, que é a de fazer *marketing* explícito para arrecadar recursos em conjunto com o *Language Adviser*. Empresas associadas as instituições podem contratar termos presentes em seus materiais, para que estas se associem a determinados termos. Quando uma organização adquire um termo, o nome desta estará relacionado com o mesmo, isto posto, toda vez que o usuário for praticar uma lição e esse termo estiver presente, o nome da organização estará logo após.

Se uma instituição tiver filiais espalhadas pelo mundo, ela tem a opção de comercializar um único termo para várias empresas distintas, isso porque quando uma empresa assina um contrato com o *Language Adviser*, entre outros atributos, ele define uma região de atuação, ou seja, em que parte do mundo ele quer que o seu nome esteja relacionado com o termo contratado. Ela pode definir que deseja se relacionar somente para um país, um estado ou uma única cidade. O *Language Adviser* usa o conceito de geocodificação para identificar qual a localização que o usuário se encontra no momento do acesso na plataforma, o uso da localização é pra definir os contratos ativos na região e fazer todo o processo de busca de termos e de injeção dos nomes das empresas nas lições (SANTOS, 2018).

Um contrato para o *Language Adviser* é a confirmação da aquisição de um ou mais

termos realizado por uma empresa com uma instituição detentora dos direitos do material vigente na plataforma. No contrato vai estar presente além dos termos adquiridos, a região em que esses termos estão vinculados e uma data de início e de término do contrato. Os termos vão estar associados a uma empresa até a data de vencimento que foi acordado, após o vencimento do contrato, esses termos estarão livres para ser contratados por uma nova empresa (SANTOS, 2018).

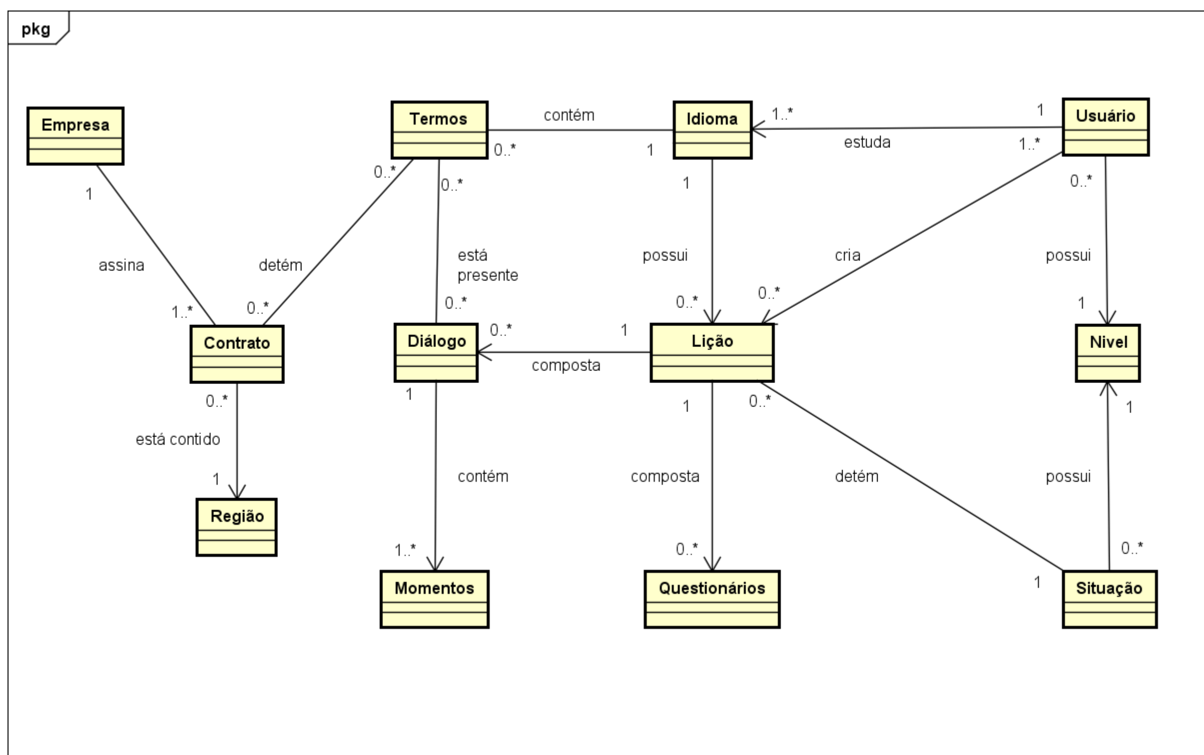
O *Language Adviser* é uma ferramenta com fundamentos de gerenciamento complexos, mas que possui funcionalidades de interação. Uma das principais funcionalidades, permite que alunos possam avaliar as lições e com isso a instituição possa obter um *feedback* do seu material e propor a criação de novos materiais que agradem mais os seus alunos.

## 3.2 Implementação

Nesta seção iremos abordar como foi feita a implementação dos requisitos da aplicação *Language Adviser* em uma API RESTful.

### 3.2.1 O Modelo

Figura 8 – Diagrama de Classe




Fonte: Autoria Própria

Para começar a explicar o modelo demonstrado no diagrama de classe da Figura 8, iremos partir do princípio de que as instituições de ensino irão cadastrar os idiomas que ela oferece na plataforma, os quais possuirão lições a serem estudadas. Uma lição é um conjunto de diálogos e questionários que é criado por usuários, professores ou supervisores das instituições, e pode ser avaliado por outros usuários, que irão estudar as mesmas. Uma lição está associada a uma situação e uma situação está associada a um nível. Uma situação representa o contexto em que uma lição está presente. O nível simboliza a dificuldade de uma situação e também a medição de aprendizado de um usuário na plataforma.

Um dos componentes de uma lição, é o diálogo, que é uma conversa entre personagens em um determinado idioma. Em um diálogo vai estar presente o nome desse diálogo, de qual lição ele se relaciona e uma lista de personagens presentes na mesma. Essa lista de personagens, serve para definir as características dos personagens e também para a identificação da cada durante o diálogo. Cada personagem possui uma fala, essa fala é considerada como um momento. Um momento indica a ordem de fala de um personagem no diálogo.

O outro componente de uma lição é um questionário, que são exercícios a serem respondidos pelos usuários. Um questionário pode ser composto por questões do tipo objetivo, subjetivo e semi-objetivo, um novo tipo criado especialmente para ampliar o número de questões que pode ser representado na plataforma. Todo questionário irá possuir um ou mais usuários com a função de supervisor, o usuário supervisor tem a responsabilidade de analisar as respostas submetidas pelos usuários comuns e fazer as respectivas correções.

Os momentos de um diálogo irão possuir frases, essas frases podem ter ou não palavras que estarão presentes na lista de termos cadastradas no banco de dados. Um termo para o *Language Adviser* é uma palavra em um determinado idioma que pode ser contratado por empresas, para vincular seu nome a determinados termos em uma região.

Uma região é uma área geográfica, que pode ser uma cidade, estado ou país, que deve ser previamente cadastrada na API, para que empresas consigam vincular contratos a essas regiões. Já o contrato é um acordo estabelecido por uma empresa com uma instituição de ensino, onde tem que ser definido a área de atuação do contrato, os termos a serem vinculados e uma data de início e vencimento.

Quando o usuário for estudar uma lição, a sua posição geográfica será capturada e enviada para API junto com o diálogo desejado para o estudo, com isso a API irá buscar se naquela região atual do usuário existe algum contrato ativo. Caso existam contratos ativos, irá buscar se existe algum termo que foi contratado e se está presente em algum dos momentos do diálogo. Caso a resposta seja afirmativa, o nome da empresa que contratou o termo na região irá aparecer ao lado do mesmo, inserido nos momentos em que aquele termo está presente; caso não exista nenhum contrato ativo ou nenhum termo presente

nos momentos, o diálogo será retornando sem modificações.

### 3.2.2 API RESTful *Language Adviser*

A implementação da API RESTful do *Language Adviser* utilizou-se da linguagem *JavaScript* na plataforma Node.js e com o auxilio do *framework Express*. Com a utilização do Node.js, permitiu a construção de uma API leve e rápida, que faz o processamento de uma grande quantidade de dados de forma superior, ao desenvolvimento de APIs em outras linguagens. O servidor HTTP do Node.js, possui configurações dificultosas, mas com o uso do *framework Express* essas configurações foram bastantes abstraídas, simplificado o seu uso. Foram utilizados dois bancos de dados NoSQL, o *ArangoDB* e o Redis. O *ArangoDB* é o banco de dados principal, responsável pelo armazenamento de todos os dados da aplicação e o Redis é o responsável pelo controle e armazenamento do cache da aplicação. A forma como ocorre a comunicação entre esses dois bancos de dados e a API será debatido na próximo tópico.

Durante o processo de implementação da API RESTful do *Language Adviser*, ocorreu uma situação que necessitou da criação de uma outra API, para não tornar a implementação mais confusa e nem centralizar muitas funcionalidades. As lições que irão compor a API *Language Adviser* é a parte fundamental do ensino e da arrecadação. Então modelar um questionário que atenda às várias formas de representação de uma questão era crucial para ser implementado.

Porém modelar um questionário e os seus diferentes tipos de questões, é muito complexo. Devido a essa complexidade e também para descentralizar os serviços da aplicação, foi criada uma outra API separado da API *Language Adviser* que é a API *questionnaire*. Com essa abordagem, tiramos toda a lógica que envolve a criação e manutenção de questionários para uma outra aplicação, com a API *Language Adviser* sendo uma cliente dessa nova aplicação.

As informações referentes aos questionários, que é armazenada no banco de dados do *Language Adviser*, é apenas o *id* do questionário e o *id* referente à lição na qual ela se relaciona. No banco de dados da API *questionnaire* é onde de fato são armazenadas as informações dos questionários e das questões que o compõem.

A API *questionnaire* é também um *web service* baseado no estilo arquitetural REST. A criação dessa nova API permite a distribuição desse serviço para outras aplicações, além da API do *Language Adviser*.

### 3.2.3 Bancos de Dados

A API *Language Adviser* se comunica com o *ArangoDB* e o Redis por meio dos módulos oficiais disponibilizados via NPM, esses módulos fornecem a comunicação entre a

API e os respectivos bancos de dados.

O *ArangoDB* é um banco de dados híbrido, que trabalha com os modelos orientado a documentos, orientado a grafos e o chave e valor, e é nesse banco de dados que são armazenadas as informações da API. Para a implementação da API *Language Adviser* foi utilizado somente o modelo orientado a documentos, pois se adequava mais às necessidades da aplicação. Foram criadas 14 coleções de documentos no *ArangoDB*, que são as seguintes coleções:

- Administrador
- Contratos
- Diálogos
- Empresas
- Estudos
- Idiomas
- Lições
- Momentos
- Nível
- Questionários
- Regiões
- Situações
- Termos
- Usuários

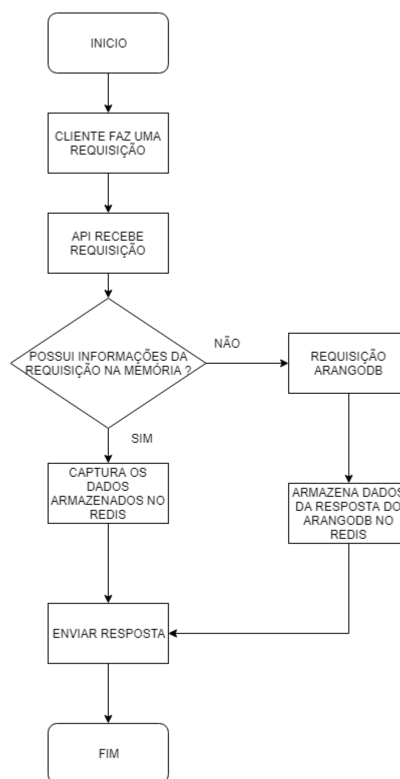
Todas essas coleções foram necessárias para a implementação dos requisitos do *Language Adviser*. O nome de cada coleção é sugestivo do que ela armazena, para facilitar a busca pelos dados.

Já o banco de dados Redis usa o modelo de dados chave e valor e trabalha com o armazenamento dos dados na memória da máquina, o que garante um ótimo desempenho. Por essa característica o Redis é a ferramenta perfeita para gerenciar e armazenar as informações temporariamente, evitando consultas desnecessárias ao banco de dados *ArangoDB*.

Quando o cliente faz uma requisição para a API, ela irá verificar com base em uma chave que é predefinida para cada requisição, se foi feita alguma requisição idêntica a essa, caso a resposta seja afirmativa ele irá devolver esses dados que estão na memória do servidor direto para o cliente, sem precisar fazer nenhuma consulta ao *ArangoDB*, reduzindo o tempo de latência e de resposta. Caso não exista nenhuma requisição idêntica à que foi feita armazenada na memória, a API irá fazer uma requisição ao *ArangoDB*, armazenar os dados da resposta no Redis e responder ao cliente que fez a requisição.

Todas as informações das respostas do *ArangoDB* têm um tempo máximo de armazenamento na memória do servidor. O tempo configurado é de 10 segundos para uma requisição que deseja listar todos os documentos e de 20 segundos para uma requisição que deseja buscar um documento especifico. Na Figura 9, temos um fluxograma que mostra como API faz quando recebe uma requisição e os passos para o envio da resposta.

Figura 9 – Fluxograma para ilustrar os passos para o envio de respostas




Fonte: Autoria Própria

### 3.2.4 Autenticação

Para proteger a API e os seus dados de uma utilização mal indevida, foi aplicado uma abordagem em *token* para autenticar os usuários que estão querendo acessar os dados

que a API fornece na sua composição. A abordagem em *token* escolhida foi o *JSON WEB TOKEN* (JWT).

O *token* do JWT possui três partes o *header, payload* e o *signature*. Para o *header* foi utilizado a criptografia padrão do JWT, ou seja, o algoritmo HS256. A composição do *payload* é formada apenas pelo *id* de cada usuário, a chave principal, e o seu tipo de acesso, ou seja, se é apenas um usuário comum ou possui permissões de administrador. Por último foi criada uma chave secreta definida no *signature*, para criar e validar todos os *tokens*. Para cada usuário é gerado um *token* diferente.

O uso do JWT é perfeito para uma API RESTful, pois toda API RESTful deve ser *stateless*, ou seja, não guardar informações de usuários por meio de sessão ou de *cookies* no servidor, em razão de que impossibilita a distribuição de serviços e a escalabilidade. Com o uso do JWT, as informações referentes sobre quem está logando na aplicação, fica no cliente, portanto, para cada requisição o cliente tem que enviar o *token* emitido pela API, para que assim a API consiga identificar quem é o usuário e determinar se ele tem permissão ou não para acessar a informação desejada.

O cliente que se comunica com a API *Language Adviser* tem que fornecer seu *token* no campo *Authorization* do cabeçalho do protocolo HTTP, para poder fazer uma requisição. Caso contrário a API irá responder uma mensagem com o código HTTP 401, ou seja, sem autorização.

Para o cliente que deseja capturar o seu *token*, só existe uma única forma que é a de fazer uma requisição para a rota login no método POST, enviando no corpo os dados de *email* e senha que foram fornecido no momento do cadastro. A resposta dessa requisição terá um atributo com o nome de *token*, o valor desse atributo é o *token* do usuário.

### 3.2.5 Escalabilidade

Uma das restrições do estilo arquitetural REST é que a API deve ser um sistema em camadas e ser *stateless*, para proporcionar o crescimento e a robustez da mesma. A API *Language Adviser* por meio do módulo *cluster* que é nativo do Node.js, implementou o balanceamento de carga, com a finalidade de ter várias máquinas trabalhando em conjunto para responder todas as requisições feitas para a API. Com o balanceamento de carga evitamos que um único processo fique responsável por responder todas as requisições e distribuímos tarefas para diferentes processos e máquinas, dessa forma, podemos atender uma maior quantidade de requisições sem apresentar nenhum gargalo e acrescentar novas máquinas no futuro sem ter nenhum problema. Essa característica de adicionar máquinas, ou seja, fazer a aplicação crescer, é justamente o conceito de escalabilidade horizontal.

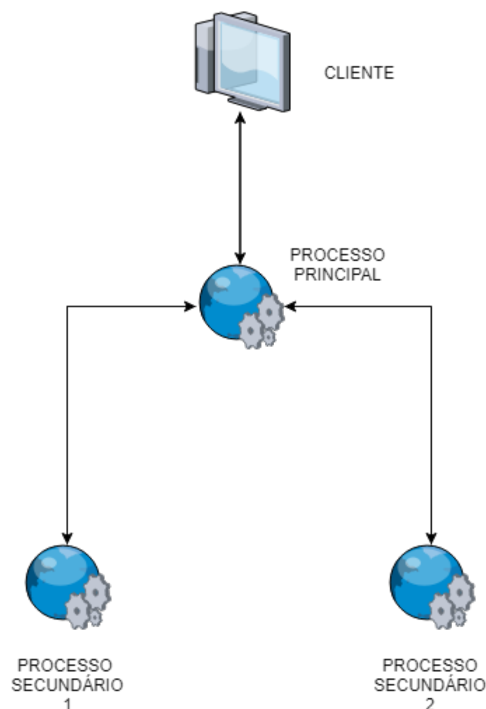
O módulo *cluster* utiliza a abordagem do algoritmo *round-robin*, onde um processo *master* atende em uma porta e distribui as tarefas a serem realizadas para os processos

*workers*. Enquanto houver *workers* vivos, o servidor continuará aceitando novas conexões (NODEJS, 2018).

O Node.js é executado por um único processo, que é uma das suas grandes vantagens em relação a outras linguagens, só que haverá ocasiões em que é preciso escalonar esse único processo, para ter um melhor aproveitamento dos recursos computacionais de uma máquina ou de um *cluster*.

Com o auxílio do módulo *cluster* foram criadas três instâncias do processo da API *Language Adviser*. A primeira instância é o processo *master*, responsável por receber todas as requisições em uma porta e escalonar as requisições para duas instâncias *workers*. São nos dois processos *workers* que ocorre o processamento das requisições. Na Figura 10, temos uma simples representação da comunicação entre os processos *workers* com o processo *master*.

Figura 10 – Balanceamento de Carga




Fonte: Autoria Próppria

### 3.2.6 Geocodificação

Um dos conceitos principais da aplicação é a parte da geocodificação, que é a tarefa de identificar a região em que o usuário se encontra. Para resolver esse problema, foi criado mais uma camada, a qual não foi preciso implementar completamente, simplesmente, a comunicamos com um serviço, que é ofertado por uma aplicação distribuída. Um cliente

para fazer *login* na API precisa passar no corpo da mensagem além do *email* e senha, a latitude e longitude, pois com esses dois atributos usando a API de geocodificação do *google maps*, conseguimos identificar com exatidão o local em que o usuário está acessando e com isso verificar os contratos que estão ativos na região em que ele está presente, no momento do acesso.

O *google maps* oferece um conjunto de APIs públicas como o de geolocalização, o de lugares, o de estradas e entre esse conjunto tem o de geocodificação. A API de geocodificação do *google maps* permite encontrar as informações de um lugar de duas formas: por meio de um endereço ou informando a latitude e longitude. A resposta que a API envia contém todos os dados que especifica com exatidão o local em que uma pessoa se encontra, mas as informações que interessam a API *Language Adviser* é somente o país, o estado e a cidade, que o usuário se encontra presente.

Para encontrar essas informações é preciso conhecer um pouco o sistema de classificações do *google maps*. Na API de geocodificação do *google maps*, um país é classificado como *country*, estado como *administrative area level 1* e cidade como *administrative area level 2*. Conhecendo as classificações de cada região, agora é preciso somente fazer uma busca no documento JSON que a API do *google maps* envia como resposta e enviar esses dados para o cliente.

A funcionalidade de capturar as coordenadas geográficas é de responsabilidade do cliente que vai consumir a API. Caso uma aplicação cliente ou um cliente não envie esses dados no corpo da mensagem, a API vai considerar o local de acesso do usuário que foi definido como padrão no momento do cadastro, caso contrário irá se conectar com a API de geocodificação do *google maps* por meio do seu módulo disponibilizado em Node.js e substituir os valores de país, estado e cidade, que foi definido no cadastro, por valores que representam a sua atual localização. Essa substituição ocorre somente na resposta da requisição de *login* e não haverá substituição nenhuma nas informações do usuário que estão armazenadas, essas informações só irão valer para o *login* feito naquele momento.

# 4 Prova de Conceito

Este trabalho resultou no desenvolvimento de um API RESTful, que seguiu todos os padrões definidos no estilo arquitetural REST e que implementou todos os fundamentos idealizados para a aplicação *Language Adviser*. Com a finalidade de validar a API, ela foi colocada em prática em uma instituição de ensino localizada na cidade de Mossoró, de fevereiro de 2018 até abril de 2018.

## 4.1 Implantando a API

Para a disponibilização da API para a instituição, foram contratados duas *Virtual Private Server* (VPS) com 512MB de memória RAM, com 20GB de armazenamento e com um endereço IP particular. Em uma VPS está funcionando um processo *master* e dois processos *workers* resultantes do algoritmo de balanceamento de carga da API, além do banco de dados Redis, já na outra VPS está hospedado o *ArangoDB* e a API *questionnaire*. Ao término de todas as configurações necessárias para que a API funcionasse sem problema, ela foi colocada *online* e pronta para ser consumida por aplicações clientes.

## 4.2 Rotas

Recursos sendo disponibilizados por *Uniform Resource Identifier* (URI) é parte da restrição da interface uniforme idealizada por Fielding (2000) e servem como ponto de entrada para o consumo da API. Para mostrar o funcionamento da API, iremos abordar nesta seção o funcionamento das principais rotas para a API. Toda a API do *Language Adviser* possui 18 recursos e 114 rotas. O detalhamento de todos esses recursos e suas respectivas rotas está disponível no apêndice A.

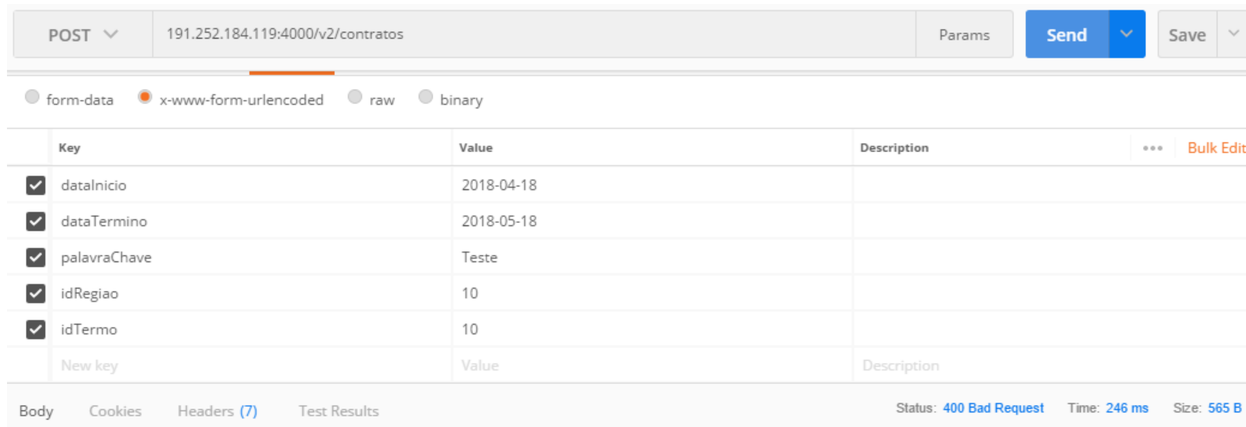
### 4.2.1 Criar Contrato

Para a criação de um contrato na API, temos que enviar uma requisição usando o método POST no recurso contratos. No corpo da mensagem temos que enviar um objeto *JSON* com os atributos de data de início e término do contrato, que vai servir para a mesma ativar e desativar os termos contratados de uma região. As datas têm que estar no formato ISO 8601, ou seja, primeiro vem o ano, depois o mês e por ultimo o dia.

Além das datas de início e término, é preciso de uma palavra chave, que significa qual palavra irá se relacionar ao termo em uma região, o *id* da empresa que vai assinar o contrato, o *id* da região, onde o contrato vai atuar, e o *id* dos termos. Uma empresa pode

colocar em único contrato vários termos, esses termos depois do contrato já estando em vigor podem ser excluídos ou ter a adição de novos.

Figura 11 – Requisição ruim




Fonte: Autoria Própria

Todos os atributos mencionados são obrigatórios para adicionar um contrato, caso esteja faltando um único atributo, como é mostrado na Figura 11, ou o nome de um atributo esteja errado, a API responderá para o cliente, um mensagem com o código de estado de número 400, ou seja, uma requisição ruim. Na Tabela 4, exibimos os detalhes da rota para adicionar um contrato.

Tabela 4 – Detalhes da rota para criar um contrato

| | |
|---|---|
| **URI** | */contratos* |
| **Método** | POST |
| **Parâmetros** | Nenhum |
| **Corpo** | dataInicio,dataTermino,palavraChave,idRegiao,idEmpresa,idTermo |

Fonte – Autoria Própria

### 4.2.2 Fornecer Diálogo

Para um cliente receber um diálogo com todas as informações e com os seus momentos, é preciso fazer uma requisição para o recurso diálogos na rota que especifique o diálogo que deseja receber, mais o */estudar*, usando o método GET. Nessa rota temos 4 parâmetros, o *id* do diálogo que vai na estrutura da rota e indica qual o diálogo que deseja recuperar, e os parâmetros de país, estado e cidade.

Com base nos parâmetros de localização que o cliente envia na rota é que a API irá buscar se existe algum contrato ativo, partindo de país, estado e cidade respectivamente. Se existir algum contrato nessas regiões, a API irá fazer uma busca se existe nos momentos

do diálogo algum termo que foi contratado e se está presente no diálogo. Caso exista, irá fazer modificação nos momentos do diálogo, inserindo o nome da empresa depois do surgimento do termo, caso não exista, os momentos serão entregues para o usuário sem nenhuma modificação.

Os parâmetros país, estado e cidade não são passados diretamente na rota, como o *id* do diálogo. Esses parâmetros são passados como uma *query string*, ou seja, logo após a URI utiliza-se um caractere especial (?) e passa os nomes dos parâmetros seguidos do seu valor. Para demonstrar a estrutura da rota, imaginemos o seguinte caso:

1. Estamos em busca das informações do diálogo e os seus momentos cujo o *id* é 40.
2. O usuário se encontra na cidade de Mossoró, no estado RN e no país Brasil.

Para esses requisitos a rota irá ter a seguinte estrutura:

*/dialogos/40/estudar?pais=Brasil&estado=RN&cidade=Mossoró*

Com a estrutura da rota e o seu funcionamento já explicado, iremos demonstrar o seu funcionamento. Criamos uma empresa fictícia chamada de supermercado Bom Demais, onde a mesma assinou um contrato que definiu a aquisição do termo *supermarket* e vai atuar somente na cidade de Mossoró. Toda vez que um usuário estiver na cidade de Mossoró e na lição aparecer o termo *supermarket*, o nome da empresa, no caso Bom Demais, irá aparecer, conforme exibe a Figura 12

Como no contrato a vinculação da empresa com o termo é somente para a região de Mossoró, então caso um usuário de outra cidade acesse a mesma lição, não irá aparecer o nome da empresa Bom Demais, como está sendo exibido na Figura 13.

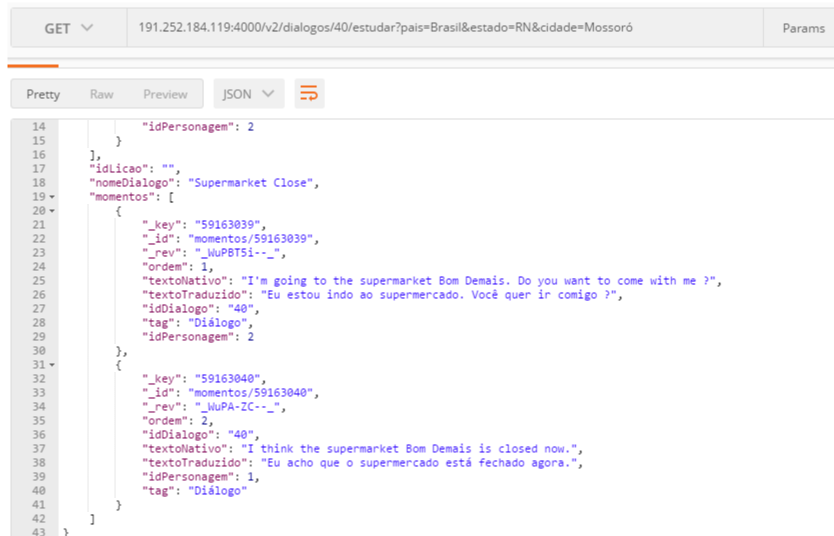
Na Tabela 5, especificamos todos os detalhes da rota de fornecer um diálogo para usuários.

Tabela 5 – Detalhes da rota de fornecimento de diálogos

| | |
|---|---|
| **URI** | */dialogos/:id/estudar* |
| **Método** | GET |
| **Parâmetros** | *ID* Diálogo,País,Estado e Cidade |
| **Corpo** | Nenhum |

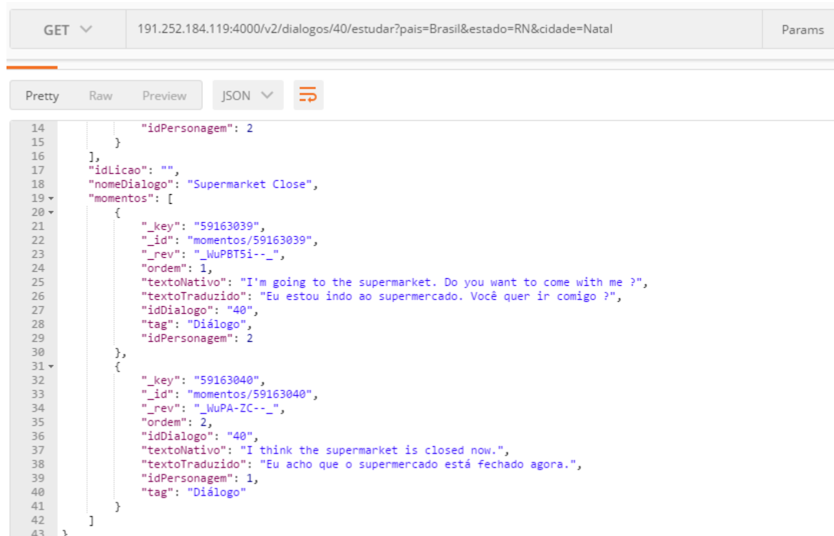
Fonte – Autoria Própria

Figura 12 – Fornecendo diálogo para usuários na região de Mossoró




Fonte: Autoria Própria

Figura 13 – Fornecendo diálogo para usuários na região de Natal

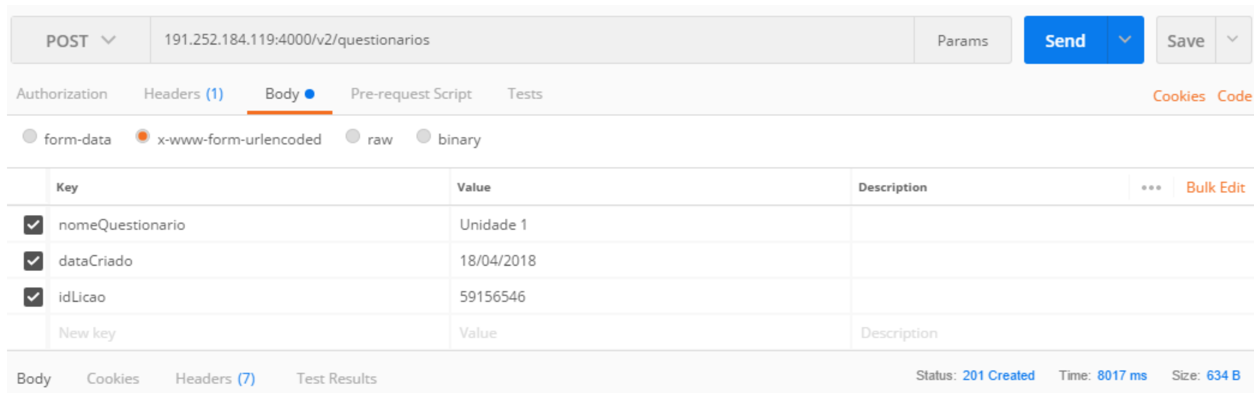



Fonte: Autoria Própria

### 4.2.3 Criar Questionário

A ação de criar um questionário faz com que a API *Language Adviser* faça uma requisição para a API *questionnaire*. Quando um cliente faz uma requisição para criar um questionário ele tem que informar no corpo da mensagem um objeto *JSON*, com os atributos nome do questionário, a data em que foi criado e o *id* da lição com que esse questionário vai se relacionar.

Figura 14 – Questionário criado com sucesso




Fonte: Autoria Própria

Quando a API *Language Adviser* recebe a requisição, ela armazena o valor do *id* da lição separadamente e prepara uma requisição para a API *questionnaire*, enviando apenas o nome do questionário e a data de criação, pois o *id* da lição não interessa para API *questionnaire* e sim para o *Language Adviser*. Depois de fazer a requisição para a API *questionnaire*, a mesma envia uma resposta contendo o *id* do questionário criado. Com o *id* do questionário, podemos relacionar um questionário armazenando em uma outra aplicação, com uma lição armazenada no banco de dados do *Language Adviser*, e enviar a resposta para o cliente que fez a requisição inicial, conforme exibe a Figura 14. Na tabela 6 temos os detalhes da rota para criar um questionário.

Tabela 6 – Detalhes da rota para criar um questionário

| | |
|---|---|
| **URI** | */questionarios* |
| **Método** | POST |
| **Parâmetros** | Nenhum |
| **Corpo** | nomeQuestionario,dataCriado,idLicao |

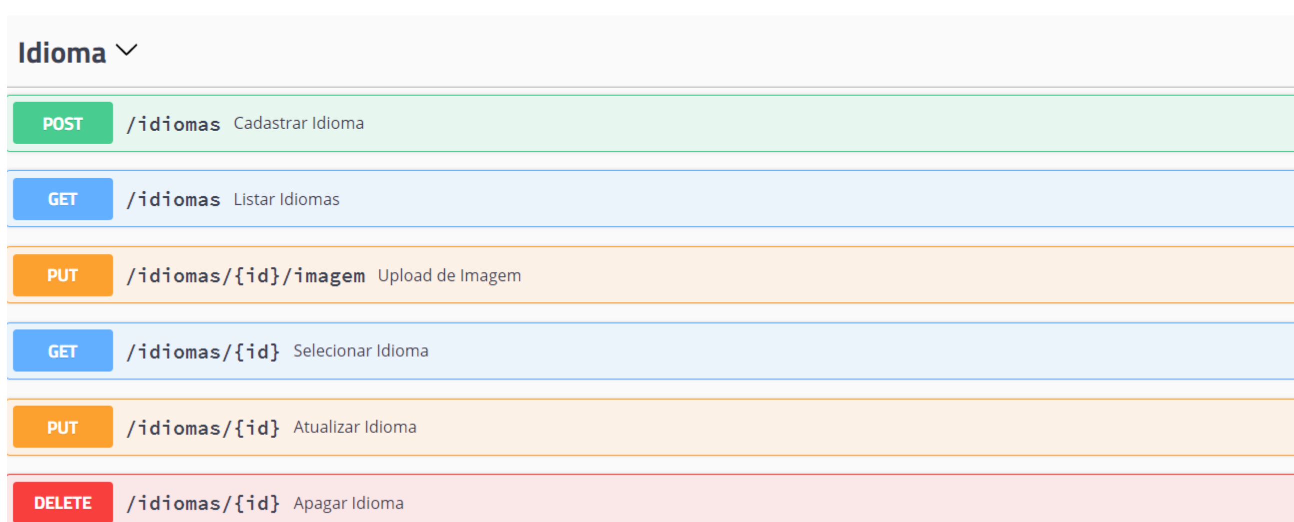
Fonte – Autoria Própria

## 4.3 Documentação

As APIs construídas no estilo arquitetural REST tem quase como obrigação oferecer uma documentação para que os desenvolvedores *front-end* consigam entender o funcionamento da mesma. Na documentação deve existir um detalhamento de cada rota, quais são os códigos de estados que uma rota pode devolver, se aceita um *content-type* com diferentes valores e muitos outros questionamentos, que sem uma documentação, seria impossível saber e que dificultaria muito a comunicação com uma API RESTful.

Todas as rotas e recursos da API do *Language Adviser* foram detalhadas com o uso da especificação *OpenAPI*. Com a criação de um documento em JSON na especificação *OpenAPI*, foi gerada uma interface *web* com a ferramenta *Swagger UI*. Essa ferramenta traduz todos os detalhes dos recursos e das rotas contidas no documento produzido pela especificação para uma interface *web*, essa interface tem funcionalidades de interação com os usuários, tornando a leitura, a compressão e o visual da documentação mais simples e bem atraente. A documentação está exposta para visualizações no endereço IP da VPS contratada. Na Figura 15, temos o exemplo de um recurso e suas respectivas rotas na interface web gerada pelo *Swagger UI* da documentação da API RESTful do *Language Adviser*.

Figura 15 – Documentação da API




Fonte: Autoria Própria

## 4.4 Aplicação Cliente

Um dos objetivos desse trabalho era proporcionar a interoperabilidade com qualquer aplicação, desenvolvida em qualquer linguagem, para justamente facilitar a criação de uma aplicação que funcionasse de forma distribuída. Portanto, depois da disponibilização da API e da documentação da mesma, tornou-se mais fácil para que aplicações clientes fossem desenvolvidas e que implementasse o modelo do *Language Adviser*, por meio das funcionalidade oferecidas pela API. Então foi desenvolvida, por um aluno do curso de Ciência da Computação da Universidade do Estado do Rio Grande do Norte (UERN), uma aplicação web, que está consumido a API e que foi construída com base na documentação criada. Com essa aplicação *web*, que vai oferecer uma interface, permitiu-se o manuseio dos dados que a API vai gerenciar, de maneira mais fácil para os alunos e para a adminis-

tração da instituição de ensino, além de provar o funcionamento da interoperabilidade da API. Essa aplicação cliente, como está se conectando com a API do *Language Adviser*, automaticamente cria uma nova camada para a aplicação *Language Adviser*, pois ela tem responsabilidades e funcionalidades distintas das outras camadas, mas que é essencial para a aplicação. Na Figura 16 temos o exemplo da interface da aplicação cliente.

Figura 16 – Interface da aplicação cliente

Fonte: Autoria Própria

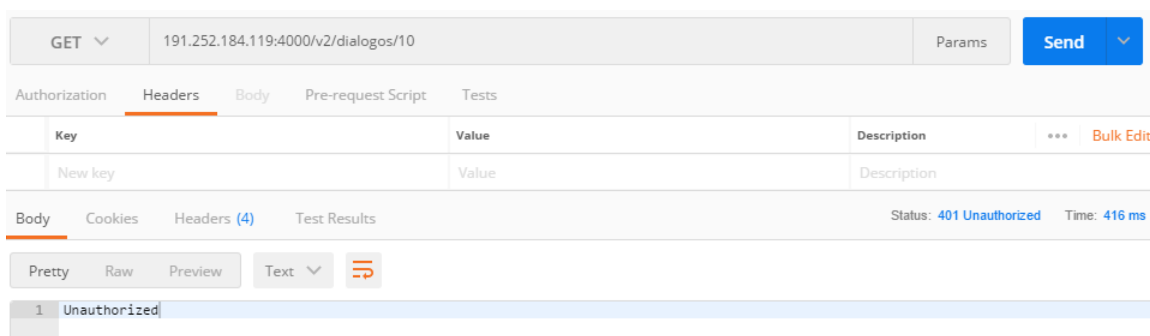
## 4.5 Resultados

Com a entrega da aplicação cliente para a instituição, a API enfim foi testada e colocada em prática. Durante três meses a API está gerenciando 108 usuários cadastrados, destes 108, apenas 2 usuários estão classificados como desativados. Como a instituição só tem a opção de ensino do idioma inglês, o único idioma cadastrado na plataforma é justamente o inglês. Foram cadastradas 30 lições, 61 diálogos que possuem um montante de 903 momentos.

A API foi totalmente desenvolvida e está plenamente funcionando na instituição, com todos os requisitos idealizados para o *Language Adivser*, porém a mesma ainda não testou a funcionalidade de oferta de termos e não criou nenhum questionário, pois a plataforma ainda está em processo de implantação em todas as filiais da instituição. Além da filial de Mossoró, ela possui outras 10 instituições espalhadas em todo o Brasil. Espera-se que ao final do processo de implantação em todas as filiais, a instituição consiga usufruir dessa funcionalidade que torna o *Language Adviser* uma plataforma única.

Com a API em ação podemos verificar se tudo que foi implementado está realmente funcionando. A segurança dos dados, que foi um dos tópicos abordados para a implementação, foi testada e comprovada sua funcionalidade. Com o uso do JWT e sua autenticação com abordagem via *token* impediu pessoas que não estão cadastradas de acessar indevidamente a API. Na Figura 17, mostramos um usuário tentando acessar a API sem informar nenhum *token*. A resposta para tipo de casos como esse, sempre vai ser o código 401, ou seja, *Unauthorized*.

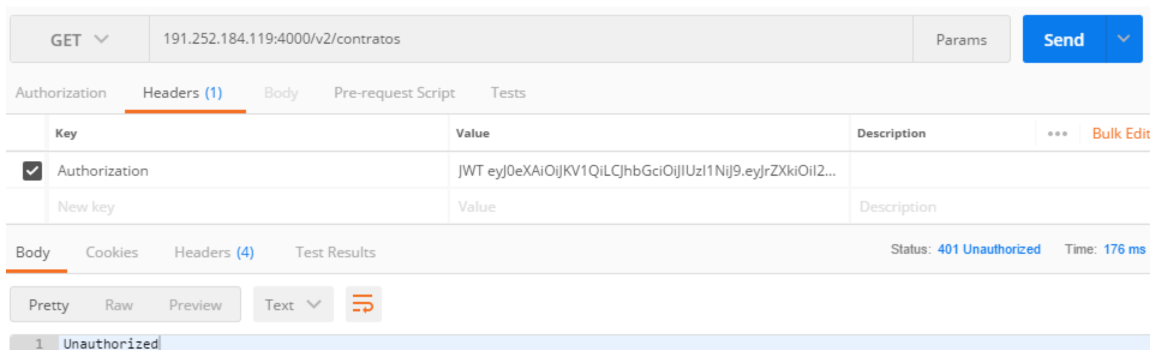
Figura 17 – Acessar rota sem *token*




Fonte: Autoria Própria

Mesmo se um usuário tiver um cadastro na API e consequentemente um *token*, ele não tem a permissão de fazer requisição para todas as rotas. Existem algumas rotas com privilégios que somente um usuário com permissão de administrador pode acessar. Todas as rotas dos recursos de contratos, termos, regiões, nível, situação e informações de outros usuários, são acessadas somente por usuários com permissões de administrador, conforme exibe a Figura 18.

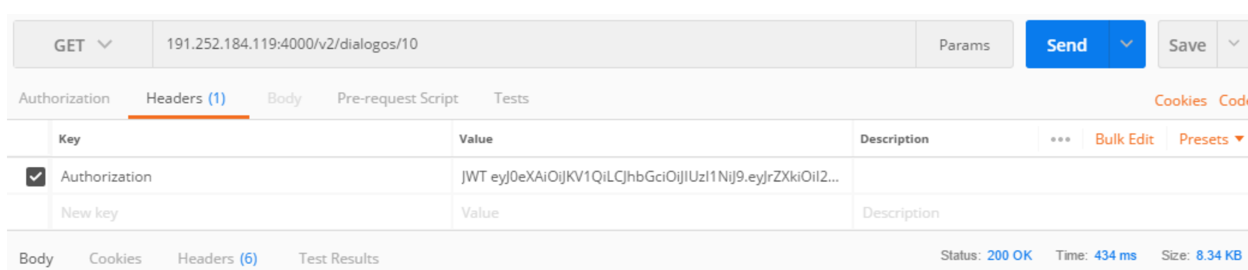
Figura 18 – Rotas com privilégios especiais




Fonte: Autoria Própria

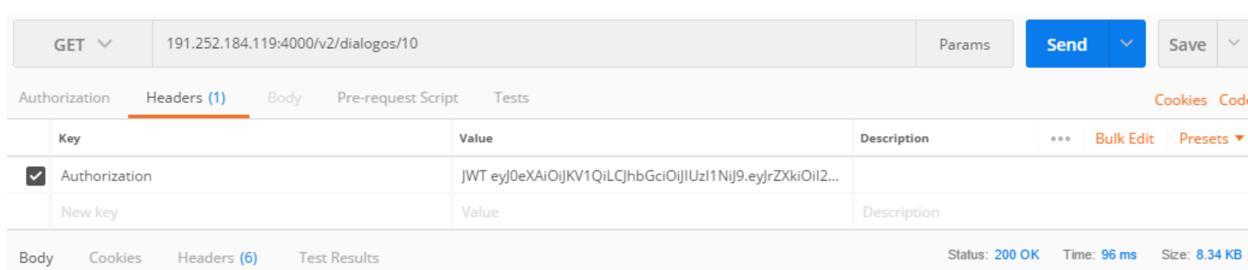
Outro tópico de extrema importância é tornar a API mais rápida e robusta, por isso a utilização do Redis. O Redis permitiu diminuir a latência e criar um cache para todas as requisições do método GET. Uma requisição para a API que não possui nenhum dado armazenado no Redis tem uma média de resposta de 500ms, enquanto que uma requisição que possui informações armazenadas no próprio, tem uma média de resposta de 200ms, uma redução de 300ms. Nas Figuras 19 e 20, temos o exemplo dos dois tipos de requisições, com os dados sem estarem armazenados e com os dados armazenados no Redis.

Figura 19 – Requisição sem informações no Redis




Fonte: Autoria Própria

Figura 20 – Requisição com informações no Redis



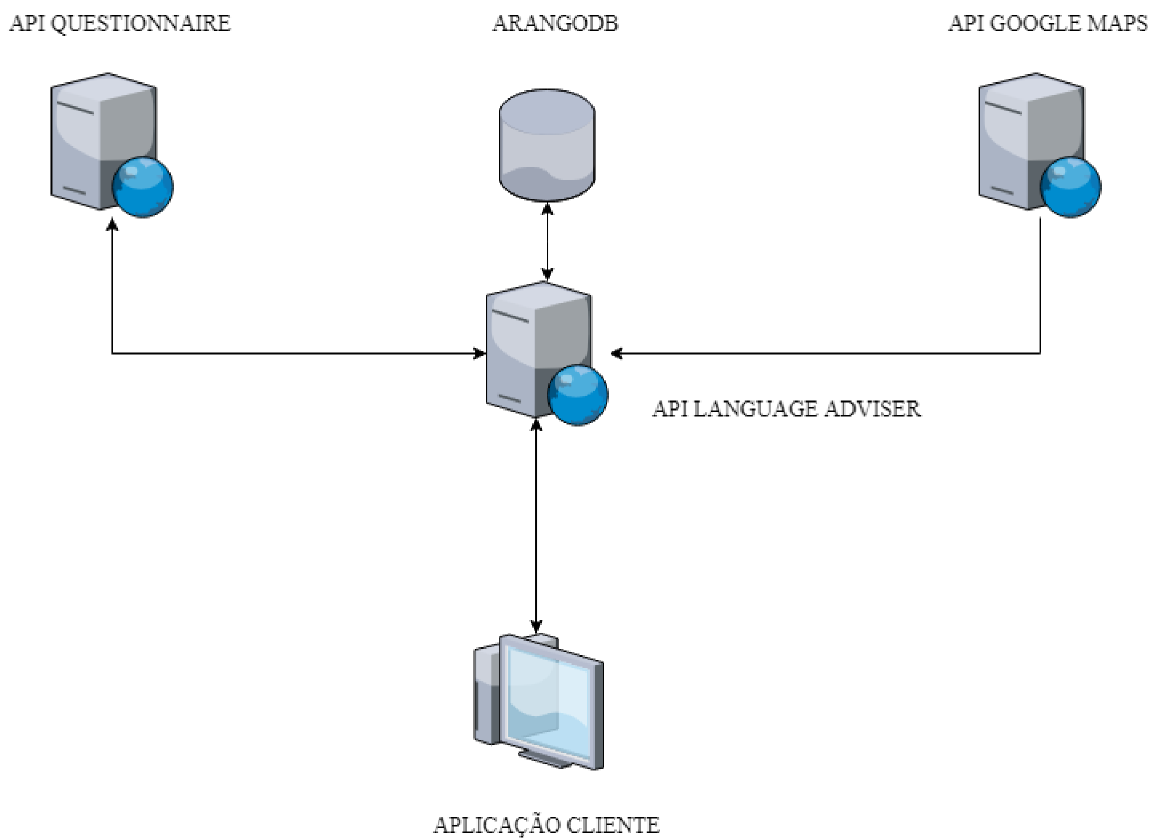

Fonte: Autoria Própria

Desde de fevereiro até o final de abril, a API foi consumida 1373 vezes, com uma crescente em relação a cada mês. O mês de fevereiro teve um total de 329 acessos, no mês de março 447 acessos e o mês abril com 620 acessos. Com esses dados provamos que, de fato, a instituição e os seus alunos estão usando a aplicação cliente e que a aplicação cliente está consumindo e usando as funcionalidades da API. Durante esse período de funcionamento, foi registrado vários picos de acessos simultâneos, o que poderia ocasionar em quedas. Com a distribuição de carga implantada na API, o serviço continuo ativo. No balanceamento de carga implementado na API, evitamos que se formem filas de requisição,

ocupação de memória e desperdício de processamento, tornando a mesma pronta para ser escalável.

Ao todo a aplicação *Language Adviser* é composta de 4 camadas distintas e distribuídas, que se comunicam como se fossem uma só. A primeira camada, que é a aplicação cliente, se comunica com um *middleware*, via HTTP, para obter dados. O *middleware* que é o objeto de estudo desse trabalho, a API RESTful do *Language Adviser*, é a segunda camada, e é a responsável por atuar como intermediário com todas as camadas, a API RESTful do *Language Adviser* integra os serviços da API *questionnaire* e a API do *Google Maps*, provendo uma interface uniforme para as aplicações clientes. Na Figura 21 ilustramos a visão completa de toda aplicação *Language Adviser*.

Figura 21 – Visão geral da aplicação




Fonte: Autoria Própria

# 5 Considerações Finais

O objetivo principal desse trabalho era a criação de uma API, baseada no estilo arquitetural REST, que implementasse os requisitos funcionais de uma aplicação, fornecesse funcionalidades que a tornassem segura, rápida e robusta, que são conceitos muito importante para qualquer *software*, promovesse a interoperabilidade e permitisse o crescimento da mesma. O uso de uma API RESTful era para mostrar, que uma API, seguindo os conceitos do REST, é a ferramenta ideal para criar aplicações distribuídas.

Então para alcançar todos os objetivos estabelecidos, foi desenvolvido uma API RESTful em Node.js que implementou todos os requisitos para a aplicação *Language Adviser*, garantiu a segurança dos dados, diminui o tempo de respostas das requisições, permitiu a interoperabilidade e tornou-se escalável. Além da criação da camada da própria API RESTful, foram criadas mais duas camadas distintas, que fornecem funcionalidades para a API, todas as camadas que foram criadas e integradas, estão distribuídas em distintas máquinas, para justamente oferecer uma maior disponibilidade. Quando a API foi finalizada e colocada *online*, permitiu a interoperabilidade com uma aplicação cliente, desenvolvida em outra linguagem, sem nenhuma relação com o desenvolvimento dessa API, e que viabilizou a criação de uma camada para usuários finais. Essa camada fim, permitiu que a API fosse validada, por meio do acompanhamento da própria em uma instituição de ensino localizada na cidade de Mossoró.

Na instituição de ensino vimos o funcionamento das quatro camadas da aplicação *Language Adviser* atuando como se fossem um só, mas na verdade estavam distribuída em várias máquinas. A camada que proporcionou a ligação com todas as outras e permitiu que a aplicação funcionassem corretamente nessa arquitetura, foi justamente a API RESTful desenvolvida nesse trabalho. Com a utilização da API RESTful conseguimos interconectar todas as camadas em torno de um só objetivo e garantir propriedades como segurança, escalabilidade e cache.

Na parceria feita com a instituição de ensino, teremos que implantar a plataforma em todas as suas filiais, consequentemente teremos um maior número de usuários ativos, o que ocasionara maiores acessos simultâneos. Mas como a API é escalável, não teremos problemas em relação a garantir a estabilidade da mesma. A única modificação será justamente o algoritmo do balanceamento de carga, que ao invés de ter dois processos terá quatro, para que atenda todas requisições sem apresentar nenhum problema.

A forma como a aplicação *Language Adviser* foi idealizada de unir o ensino e a publicidade em uma só plataforma, o tornava muito complicado de se implementar como uma aplicação centralizada, por isso, era o exemplo perfeito, para demonstrar os

benefícios, da utilização de uma API. Existiam restrições e preocupações que só podia ser atendida com a criação de uma API, seguindo os princípios estabelecidos no REST, pois a descentralização e a distribuição de serviços, favoreceu a implementação de funcionalidades importantes para o *Langauge Adviser*, além de tornar a mesma uma aplicação distribuída.

Este trabalho conseguiu alcançar todos os seus objetivos estabelecidos, pois com a construção da API, no estilo arquitetural REST, provamos que de fato ela é a ferramenta ideal para promover serviços de forma descentralizadas, alcançar interoperabilidade e permitir a implementação de requisitos para tornar uma aplicação distribuída. Nessa API foram implementados propriedades que foram e serão necessárias para o futuro da aplicação, essas propriedades foram implementados de forma mais fácil, porque com a construção de uma API, podemos uniformizar um estilo de comunicação para integrar distintas camadas de uma aplicação.

Como perspectivas futuras, poderíamos criar uma API *GraphQL* do *Language Adviser* e comparar com a API RESTful desenvolvida nesse trabalho. O *GraphQL* é um estilo arquitetural desenvolvido pelo *Facebook*, para promover a consulta de dados, de forma mais simples, rápida e direta, que em relação ao REST. Ao fazer essa comparação poderíamos traçar em quais casos é melhor utilizar o estilo arquitetural REST ou o *GraphQL*.

# Referências

ABADI, D. J.; BONCZ, P. A.; HARIZOPOULOS, S. Column-oriented database systems. *Proceedings of the VLDB Endowment*, VLDB Endowment, v. 2, n. 2, p. 1664–1665, 2009.

ABERNETHY, M. Just what is node. js. *Luettu*, v. 16, p. 2012, 2011.

ANGLES, R.; GUTIERREZ, C. Survey of graph database models. *ACM Computing Surveys (CSUR)*, ACM, v. 40, n. 1, p. 1, 2008.

ARANGODB. *ArangoDB Documentation*. 2018. Acessado em 23/03/2018. Disponível em: <https://docs.arangodb.com/3.3/Manual/index.html>.

BARRETO, C. Um modelo computacional para integração de problemas de otimização utilizando banco de dados orientados a grafos. 2017.

BAX, M. P.; LEAL, G. J. Serviços web e a evolução dos serviços em ti. *DataGramaZero: Revista de Ciência da Informação. Rio de Janeiro*, v. 2, n. 2, 2001.

BOCHI, J. da S. *Programaçao Paralela no Banco de Dados Chave-Valor Redis*. Tese (Doutorado) — PUC-Rio, 2015.

BRITO, R. W. Bancos de dados nosql x sgbds relacionais: análise comparativa. *Faculdade Farias Brito e Universidade de Fortaleza*, 2010.

BURKE, T. C.; REDDY, A. S.; SRINIVASAN, A. *Technique for finding rest resources using an n-ary tree structure navigated using a collision free progressive hash*. [S.l.]: Google Patents, 2010. US Patent 7,774,380.

CARLSON, J. L. *Redis in action*. [S.l.]: Manning Publications Co., 2013.

CAVALEIRO, A. F. S. Estilo arquitetural rest para criação de web services restful. 2013.

CIRIACO, D. *O que é API?* 2009. Acessado em 10/04/2018. Disponível em: <http://www.tecmundo.com.br/programacao/1807-o-que-e-api-.htm>.

CONCEICAO, H. M. R. S. d. Plataforma de gestão de importação. In: *Plataforma de Gestão de Importação*. [S.l.: s.n.], 2015.

COSTA, B. et al. Evaluating a representational state transfer (rest) architecture: What is the impact of rest in my architecture? In: IEEE. *Software Architecture (WICSA), 2014 IEEE/IFIP Conference on*. [S.l.], 2014. p. 105–114.

COULOURIS, G. F.; DOLLIMORE, J.; KINDBERG, T. *Distributed systems: concepts and design*. [S.l.]: Pearson Education, 2012.

DIANA, M. D.; GEROSA, M. A. Nosql na web 2.0: Um estudo comparativo de bancos não-relacionais para armazenamento de dados na web 2.0. In: *IX Workshop de Teses e Dissertações em Banco de Dados*. [S.l.: s.n.], 2010. v. 9.

FERREIRA, F. et al. Especificação de interfaces aplicacionais rest. *Actas do 9o Encontro Nacional de Informática, INFORUM*, 2017.

FIELDING, R. T. *Architectural styles and the design of network-based software architectures*. Tese (Doutorado) — University of California, Irvine, 2000.

FIELDING, R. T. Rest apis must be hypertext-driven. *Untangled musings of Roy T. Fielding*, p. 24, 2008.

FOWLER, M. Richardson maturity model: steps toward the glory of rest. *Online at http://martinfowler. com/articles/richardsonMaturityModel. html*, p. 24–65, 2010.

GALVAO, J. N.; SOARES, F. A.; KAI, P. M. Modelo de api rest para integracao de sistemas biométricos agn osticos. 2017.

GURUGÉ, A. *Web services: theory and practice*. [S.l.]: Elsevier, 2004.

HAHN, E. *Express in Action: Writing, building, and testing Node. js applications*. [S.l.]: Manning Publications,, 2016.

HOLANDA, M.; SOUZA, J. A. Query languages in nosql databases. *Handbook of Research on Innovative Database Query Processing Techniques*, IGI Global, p. 415, 2015.

HUGHES-CROUCHER, T.; WILSON, M. Up and running with node. js. *Up and Running*, O'Reilly, v. 1, 2012.

JWT. *JWT Documentation*. 2018. Acessado em 24/03/2018. Disponível em: <https://jwt.io>.

LÓSCIO, B. F.; OLIVEIRA, H. R. d.; PONTES, J. C. d. S. Nosql no desenvolvimento de aplicações web colaborativas. *VIII Simpósio Brasileiro de Sistemas Colaborativos*, v. 10, n. 1, p. 11, 2011.

MASSE, M. *REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces*. [S.l.]: "O'Reilly Media, Inc.", 2011.

MONTANHEIRO, L. S.; CARVALHO, A. M. M.; RODRIGUES, J. A. Utilização de json web token na autenticação de usuários em apis rest. 2016.

MORAES, J. et al. Web services. *Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS), Rio Grande do Sul, Brasil*, 2013.

MORO, T. D.; DORNELES, C.; REBONATTO, M. T. Web services ws-* versus web services rest. *Revista de Iniciação Científica*, v. 11, n. 1, 2009.

NGOLO, M. A. F. *Arquitectura orientada a serviços REST para laboratórios remotos*. Tese (Doutorado) — FCT-UNL, 2009.

NODEJS. *NodeJS Documentation Cluster*. 2018. Acessado em 26/03/2018. Disponível em: <https://nodejs.org/docs/latest-v8.x/api/cluster.html>.

OLIVEIRA, F. R.; CURA, L. M. del V. Avaliação do desempenho de gerenciadores de bancos de dados multi modelo em aplicações com persistência poliglota. 2017.

OLIVEIRA, P. H. C. Desenvolvimento de um gerador de api rest seguindo os principais padrões da arquitetura. 2015.

OPENAPI. *OpenAPI Documentation*. 2018. Acessado em 03/04/2018. Disponível em: <https://swagger.io/specification>.

PORTO, I. O. et al. Padrões e diretrizes arquiteturais para escalabilidade de sistemas. Universidade Federal de Uberlândia, 2009.

PULUCENO, T. V. Estudo de caso sobre uma api rest em node. js. 2012.

RIBEIRO, M.; FRANCISCO, R. Web services rest conceitos, análise e implementação. *Educação, Tecnologia e Cultura-ETC*, n. 14, 2016.

RICHARDSON, L. *The Maturity Heuristic*. [S.l.]: Justice Will take us Millions of Intricate Moves. Act, 2008.

RICHARDSON, L.; RUBY, S. *RESTful web services*. [S.l.]: "O'Reilly Media, Inc.", 2008.

ROBINSON, I.; WEBBER, J.; EIFREM, E. *Graph databases: new opportunities for connected data*. [S.l.]: "O'Reilly Media, Inc.", 2015.

SAMPAIO, P. J.; KNOP, I. de O. Desempenho de aplicações web: Um estudo comparativo utilizando o software redis. *Caderno de Estudos em Sistemas de Informação*, v. 2, n. 2, 2015.

SANTOS, W. S. J. Language adviser: Aplicacao multiplataforma para o apoio do ensino de lingua estrangeira. 2018.

SEVERANCE, C. Javascript: Designing a language in 10 days. *Computer*, IEEE, v. 45, n. 2, p. 7–8, 2012.

SHETH, A. P. Changing focus on interoperability in information systems: from system, syntax, structure to semantics. In: *Interoperating geographic information systems*. [S.l.]: Springer, 1999. p. 5–29.

SOARES, B. E.; BOSCARIOLI, C. Modelo de banco de dados colunar: Características, aplicações e exemplos de sistemas. 2012.

SOUSA, B. A. d. Proveniência de dados de workflows de bioinformática usando o banco de dados no sql arangodb. 2016.

SOUSA, F. P. de. Criação de framework rest/hateoas open source para desenvolvimento de apis em nodejs. 2015.

W3C. *W3C*. 2018. Acessado em 02/05/2018. Disponível em: <https://www.w3.org/TR/ws-arch/>.

# APÊNDICE A – Recursos e Rotas

As rotas são os pontos de acesso de uma aplicação cliente que deseja se conectar a uma API REST. As rotas são classificadas em recursos, os recursos são os principais tópicos disponibilizados por um API RESTful. Neste apêndice será descrito todos os recursos e as rotas que foram criadas para a API RESTful do *Language Adviser*.

Tabela 1 – Rotas do Login

| URI | MÉTODO | DESCRIÇÃO |
|---|---|---|
| */login* | POST | Faz login na API. |
| */login/redefinir* | PUT | Redefine a senha de usuário. |

Fonte – Autoria Própria

Tabela 2 – Rotas do Administrador

| URI | MÉTODO | DESCRIÇÃO |
|---|---|---|
| */administradores* | POST | Cadastra um administrador. |
| */administradores* | GET | Lista todos os administradores. |
| */administradores/:id* | GET | Busca um administrador pelo ID. |
| */administradores/:id* | PUT | Edita os dados de um administrador. |
| */administradores/:id/imagem* | PUT | *Upload* da imagem de perfil. |
| */administradores/:id* | DELETE | Deleta um administrador. |

Fonte – Autoria Própria

Tabela 3 – Rotas do Contrato

| URI | MÉTODO | DESCRIÇÃO |
|---|---|---|
| */contratos* | POST | Cadastra um contrato. |
| */contratos* | GET | Lista todos os contratos . |
| */contratos/:id* | GET | Busca um contrato pelo ID. |
| */contratos/:id* | PUT | Edita as informações de um contrato. |
| */contratos/:id* | DELETE | Deleta um contrato. |
| */contratos/ativo* | GET | Lista todos os contratos ativos. |
| */contratos/expirado* | GET | Lista todos os contratos expirados. |
| */contratos/:idEmpresa* | GET | Lista de contratos de uma empresa. |
| */contratos/:idTermos* | GET | Lista todos os contratos de um termo. |
| */contratos/:idRegiao* | GET | Lista todos os contratos de uma região. |
| */contratos/adicionarTermo* | PUT | Adiciona um termo em um contrato. |
| */contratos/removerTermo* | PUT | Remove um termo em um contrato. |

Fonte – Autoria Própria

Tabela 4 – Rotas do Diálogo

| URI | MÉTODO | DESCRIÇÃO |
|---|---|---|
| */dialogos* | POST | Cadastra um diálogo. |
| */dialogos* | GET | Lista todos os diálogos. |
| */dialogos/:id* | GET | Busca um diálogo pelo ID. |
| */dialogos/:id* | PUT | Edita as informações de um diálogo. |
| */dialogos/:id* | DELETE | Deleta um diálogo. |
| */dialogos/:id/estudar* | PUT | Diálogo com todos os momentos. |
| */dialogos/licao/:idLicao* | GET | Lista todos os diálogos de uma lição. |
| */dialogos/:id/audio* | PUT | *Upload* do áudio para o diálogo. |

Fonte – Autoria Própria

Tabela 5 – Rotas da Empresa

| URI | MÉTODO | DESCRIÇÃO |
|---|---|---|
| */empresas* | POST | Cadastra uma empresa. |
| */empresas* | GET | Lista todas as empresas. |
| */empresas/:id* | GET | Busca um empresa pelo ID. |
| */empresas/:id* | PUT | Edita as informações de uma empresa. |
| */empresas/:id* | DELETE | Deleta uma empresa. |

Fonte – Autoria Própria

Tabela 6 – Rotas de Estudo

| URI | MÉTODO | DESCRIÇÃO |
|---|---|---|
| */estudos* | POST | Cadastra um idioma para um usuário. |
| */estudos* | GET | Lista todos os idiomas estudados. |
| */estudos/:id* | GET | Busca o idioma de um usuário. |
| */estudos* | PUT | Deleta um idioma de estudo do usuário. |
| */estudos/:id* | DELETE | Exclui todos os idiomas do usuário. |

Fonte – Autoria Própria

Tabela 7 – Rotas do Idioma

| URI | MÉTODO | DESCRIÇÃO |
|---|---|---|
| */idiomas* | POST | Cadastra um idioma. |
| */idiomas* | GET | Lista todos os idiomas. |
| */idiomas/:id* | GET | Busca um idioma pelo ID. |
| */idiomas/:id* | PUT | Edita as informações de um idioma. |
| */idiomas/:id/imagem* | PUT | *Upload* de imagem para o idioma. |
| */idiomas/:id* | DELETE | Deleta um idioma. |

Fonte – Autoria Própria

Tabela 8 – Rotas das Lições

| URI | MÉTODO | DESCRIÇÃO |
|---|---|---|
| */licoes* | POST | Cadastra uma lição. |
| */licoes* | GET | Lista todas as lições |
| */licoes/:id* | GET | Busca uma lição pelo ID. |
| */licoes/:id* | PUT | Edita as informações de uma lição. |
| */licoes/:id* | DELETE | Deleta um lição. |
| */licoes/usuarios/:idUsuario* | GET | Lista todas as lições de um usuário. |
| */licoes/situacao/:idSituacao* | GET | Lista todas as lições de um situação. |
| */licoes/avaliacao* | PUT | Avalia uma lição. |

Fonte – Autoria Própria

Tabela 9 – Rotas de Momentos

| URI | MÉTODO | DESCRIÇÃO |
|---|---|---|
| */momentos* | POST | Cadastra um momento. |
| */momentos/:id* | PUT | Edita as informações de um momento. |
| */momentos/:id/imagem* | PUT | *Upload* de imagem para o momento. |
| */momentos/:id* | DELETE | Deleta um momento. |

Fonte – Autoria Própria

Tabela 10 – Rotas dos Níveis

| URI | MÉTODO | DESCRIÇÃO |
|---|---|---|
| */niveis* | POST | Cadastra um nível. |
| */niveis* | GET | Lista todas os níveis. |
| */niveis/:id* | GET | Busca uma nível pelo ID. |
| */niveis/:id* | PUT | Edita as informações de um nível. |
| */niveis/:id/imagem* | PUT | *Upload* de imagem para o nível. |
| */niveis/:id* | DELETE | Deleta um nível. |

Fonte – Autoria Própria

Tabela 11 – Rotas das Regiões

| URI | MÉTODO | DESCRIÇÃO |
|---|---|---|
| */regioes* | POST | Cadastra uma região. |
| */regioes* | GET | Lista todas as regiões. |
| */regioes/:id* | GET | Busca uma região pelo ID. |
| */regioes/:id* | PUT | Edita as informações de um região. |
| */regioes/:id* | DELETE | Deleta uma região. |

Fonte – Autoria Própria

Tabela 12 – Rotas das Questões

| URI | MÉTODO | DESCRIÇÃO |
|---|---|---|
| */questoes* | POST | Cadastra uma questão. |
| */questoes/:id* | GET | Busca uma questão pelo ID. |
| */questoes/:id* | PUT | Edita as informações de uma questão. |
| */questoes/:id* | DELETE | Deleta uma questão. |

Fonte – Autoria Própria

Tabela 13 – Rotas dos Questionários

| URI | MÉTODO | DESCRIÇÃO |
|---|---|---|
| */questionarios* | POST | Cadastra uma questionário. |
| */questionarios/:id* | GET | Busca um questionário pelo ID. |
| */questionarios/licao/:idLicao* | GET | Lista todos os questionários da lição. |
| */questionarios/:id* | PUT | Edita as informações do questionário. |
| */questionarios/:id* | DELETE | Deleta um questionário. |

Fonte – Autoria Própria

Tabela 14 – Rotas das Respostas

| URI | MÉTODO | DESCRIÇÃO |
|---|---|---|
| */respostas* | POST | Cadastra uma resposta. |
| */respostas/:id* | GET | Busca uma reposta pelo ID. |
| */respostas/:id* | PUT | Edita as informações de uma resposta. |

Fonte – Autoria Própria

Tabela 15 – Rotas de Study

| URI | MÉTODO | DESCRIÇÃO |
|---|---|---|
| */studies* | POST | Cadastrar *study*. |
| */studies/usuarios/:idUsuario* | GET | Busca todos os *study* de um usuário. |
| */studies/:id/respostas* | GET | Listar todos as respostas de um *study*. |
| */studies/:id* | GET | Busca um *study* pelo ID. |
| */studies/:id* | PUT | Edita as informações de um *study*. |

Fonte – Autoria Própria

Tabela 16 – Rotas das Situações

| URI | MÉTODO | DESCRIÇÃO |
|---|---|---|
| */situacoes* | POST | Cadastra uma situação. |
| */situacoes* | GET | Lista todas as situações. |
| */situacoes/:id* | GET | Busca uma situação pelo ID. |
| */situacoes/idioma/:idIdioma* | GET | Lista todas as situações de um idioma. |
| */situacoes/nivel/:idNivel* | GET | Lista todas as situações de um nível. |
| */situacoes/:id* | PUT | Edita as informações de uma situação. |
| */situacoes/:id/imagem* | PUT | *Upload* de imagem da situação. |
| */situacoes/:id* | DELETE | Excluir uma situação. |

Fonte – Autoria Própria

Tabela 17 – Rotas dos Termos

| URI | MÉTODO | DESCRIÇÃO |
|---|---|---|
| */termos* | POST | Cadastra uma termo. |
| */termos* | GET | Lista todos os termos. |
| */termos/:id* | GET | Busca um termo pelo ID. |
| */termos/idioma/:idIdioma* | GET | Lista todas os termos de um idioma. |
| */termos/:id* | PUT | Edita as informações de um termo. |
| */termos/:id* | DELETE | Excluir um termo. |

Fonte – Autoria Própria

Tabela 18 – Rotas do Usuário

| URI | MÉTODO | DESCRIÇÃO |
|---|---|---|
| */usuarios* | POST | Cadastra um usuário. |
| */usuarios* | GET | Lista todos os usuários. |
| */usuarios/:id* | GET | Busca um usuário pelo ID. |
| */usuarios/:id/avaliacao* | GET | Avalia um usuário. |
| */usuarios/ranking* | GET | Ranking dos usuários. |
| */usuarios/:id/questionarios* | GET | Lista os questionários do supervisor. |
| */usuarios/nivel/:idNivel* | GET | Lista os usuários de um nível. |
| */usuarios/ativos* | GET | Lista os usuários ativos. |
| */usuarios/desativos* | GET | Lista os usuários desativados. |
| */usuarios/:idU/questionarios/:idQ* | PUT | Atribui um questionário ao usuário. |
| */usuarios/:id/supervisor* | PUT | Atribui a função de supervisor. |
| */usuarios/:id* | PUT | Edita as informações de um usuário. |
| */usuarios/:id/imagem* | PUT | *Upload* da imagem de perfil. |
| */usuarios/:idU/questionarios/idQ* | DELETE | Deleta o questionário do supervisor. |
| */usuarios/:id* | DELETE | Deleta um usuário. |

Fonte – Autoria Própria