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

JOÃO ANDRÉ NOBERTO DE SOUZA

UMA API REST PARA CONSUMO DE DADOS GEORREFERENCIADOS UTILIZANDO ARANGODB

MOSSORÓ - RN

2018

João André Noberto de Souza

Uma API REST Para Consumo de Dados Georreferenciados Utilizando ArangoDB

Monografia apresentada como requisito parcial para obtenção do título de Bacharel em Ciência da Computação, de acordo com o Projeto Pedagógico do Curso de Bacharelado em Ciência da Computação da Universidade do Estado do Rio Grande do Norte - UERN.

Orientador: Prof. Me. Antônio Oliveira Filho

Mossoró - RN

2018

S729a Souza, João André Noberto de
Uma API REST Para Consumo de Dados Georreferenciados Utilizando ArangoDB/ João André Noberto de Souza. – Mossoró, 2018.
68 p.

Orientador: Prof. Me. Antônio Oliveira Filho.
Monografia (Graduação) – Universidade do Estado do Rio Grande do Norte, Ciência da Computação, Mossoró, RN, 2018.

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

Ficha catalográfica gerada pelo Sistema Integrado de Bibliotecas e Diretoria de Informatização (DINF) - UERN, com os dados fornecidos pelo(a) autor(a).

João André Noberto de Souza

Uma API REST Para Consumo de Dados Georreferenciados Utilizando ArangoDB

Monografia apresentada como requisito parcial para obtenção do título de Bacharel em Ciência da Computação, de acordo com o Projeto Pedagógico do Curso de Bacharelado em Ciência da Computação da Universidade do Estado do Rio Grande do Norte - UERN.

Aprovada em: 20/06/2018

Banca Examinadora

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

Prof. ALYSSON MENDES DE OLIVEIRA
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

Prof. Me. Antônio Oliveira Filho (Orientador)

Especialmente dedicado a todos aqueles que veem na ciência o triunfo da nossa espécie.

AGRADECIMENTOS

Primeiramente, agradeço a minha família por sempre acreditar e investir em meus potenciais. Especialmente à minha mãe, Ana Celeste, por dedicar parte de sua vida aos meus sonhos, ao meu tio, Marcos Noberto, por compartilhar comigo os ensinamentos de um verdadeiro pai, aos meus avós, João Raimundo e Maria José, por ajudarem a trilhar os caminhos ao longo de minha vida, fazendo de mim o homem que sou hoje. A minha namorada, Gisélia, por todo o incetivo e força que me deu em todos os momentos e a todos os familiares que contribuíram para o meu crescimento pessoal.

A todos do Departamento de Informática da UERN, por compartilharem sua sabedoria e por contribuírem para o meu crescimento profissional e intelectual, em especial ao professor Alysson Mendes por todo o seu esforço em compartilhar o espírito de um legítimo cientista da computação, ao professor Marcelino Pereira por acreditar em meu potencial e principalmente ao meu mentor e professor orientador, Antônio Oliveira, pela confiança e reconhecimento dado a mim ao longo da graduação, além de sua imensa dedicação e ensinamentos, que levarei de maneira sábia durante toda a minha carreira.

A todos os professores com quem tive a oportunidade de aprender ao longo de minha vida.

Aos amigos que adquiri durante minha permanência na UERN, vocês são uma parte essencial dessa conquista.

Gostaria de agradecer àqueles que, de algum modo, contribuíram com a evolução da ciência, nos proporcionando a alegria de explorar novos cenários e moldar um mundo aonde as futuras gerações poderão viver melhor. Por fim, obrigado aos que sonham com o amanhã, aos que olham além do horizonte, aos que acreditam que a história do universo é a nossa história.

“If you wish to make an apple pie from scratch, you must first invent the universe."

(Carl Sagan)

RESUMO

O georreferenciamento, dentre suas inúmeras aplicações, facilita o controle sobre o registro de imóveis, pois fornece informações necessárias para evitar a sobreposição de títulos de propriedade. Todo esse processo é realizado através do uso de SIGs e, embora tenham informações precisas, eles são limitados quando se trata da manipulação de informações em diferentes aplicações. A necessidade de integração entre sistemas criados em diferentes linguagens é cada vez mais comum em grandes empresas, principalmente após a explosão da internet. As aplicações tiveram que se adaptar a uma nova realidade, com isso, o surgimento de novas tecnologias tornou popular o uso de Web Services, permitindo oferecer maneiras de integrar diferentes sistemas, modularizar serviços, unificar e possibilitar a integração e o consumo de informações. Portanto, este trabalho se concentra no desenvolvimento e implementação de uma API, com base no estilo arquitetural REST. Assim, uma API REST foi implementada com a ajuda do framework Node.js e Express, interagindo com um banco de dados NoSQL, o ArangoDB.

Palavras-chaves: Estilo Arquitetural REST, Georreferenciamento, Web Services.

ABSTRACT

Georeferencing, among its many applications, can easily control real estate registration, as it provides information needed to avoid land titles overlay. All this process is handled by a GIS and although they have pretty much accurate information, they are limited when it comes to the manipulation of information over external applications. The need of integration between systems created in different languages is commonly increasing in large companies, especially after the explosion of the internet. The applications had to adapt themselves to a new reality, the emergence of new technologies made the use of Web Services very popular, allowing to offer ways to integrate different systems, modularize services, unify and enable integration and consumption of information. Therefore, this work focuses on developing and implementing an API, based on the REST architectural style. Thus, a REST API was implemented with the use of Node.js and Express framework, interacting with a NoSQL database, ArangoDB.

Key-words: Architectural REST, Georeferencing, Web Services.

LISTA DE FIGURAS

LISTA DE ABREVIATURAS E SIGLAS

ACID	Atomic, Consistent, Isolated, Durable
API	Application Programming Interface
AQL	ArangoDB Query Language
BASE	Basic Availability, Soft-state, Eventual Consistency
CLI	Command Line Interface
CRUD	Create, Read, Update, Delete
GIS	Geographic Information System
HTTP	Hypertext Transfer Protocol
JSON	JavaScript Object Notation
LORDI	Laboratório de Redes e Sistemas Distribuídos
NoSQL	Not Only SQL
NPM	Node Package Manager
RDBMS	Relational Database Management System
REST	Representational State Transfer
SGBD	Sistema de Gerenciamento de Banco de Dados
SGBDR	Sistema Gerenciador de Banco de Dados Relacional
SIG	Sistema de Informação Geográfica
SQL	Structured Query Language
UERN	Universidade do Estado do Rio Grande do Norte
URI	Universal Resource Identifier
URL	Universal Resource Locator
WGS	World Geodetic System
W3C	World Wide Web Consortium
WWW	World Wide Web
XML	eXtensible Markup Language

SUMÁRIO

1 INTRODUÇÃO

Do ponto de vista social, o georreferenciamento é uma ferramenta que facilita o controle mais eficaz sobre os registros dos imóveis rurais porque fornece as informações necessárias para evitar títulos de terra sobrepostos, minimizando as dúvidas, os litígios e as grilagens no meio rural. (MENZORI, 2017)

Segundo Menzori (2017), o ato de georreferenciar significa determinar a posição de pontos, linhas e polígonos usando coordenadas referidas a um sistema único mundial. O georreferenciamento nos permite representar pontos na superfície geográfica através de suas coordenadas.

Apesar de possuirem informações precisas e que seguem notações específicas para representar estruturas de dados geográficos, os *SIG*s podem apresentar-se limitados quando se trata da manipulação de informações em suas bases de dados, dificultando o compartilhamento de informações e interoperabilidade com outras aplicações.

A necessidade de integração entre sistemas criados em diferentes linguagens é encontrada cada vez mais nas grandes empresas. 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 consumo de informações. Levando em consideração que o desenvolvimento de um *Web Service* para auxílio na comunicação e troca de mensagens entre os dados de *SIG*s utilizará o protocolo *HTTP* (*Hypertext Transfer Protocol*), pode-se adotar um padrão que utiliza todos os recursos que esse protocolo oferece, usando um dos padrões mais utilizados atualmente, o *REST* (*Representational State Transfer*). (OLIVEIRA, 2014)

Com a utilização do estilo arquitetural *REST*, pode-se criar facilmente uma API capaz de se comunicar com outros clientes. Ao implementar uma API que será consumida através do protocolo HTTP, temos, na verdade, um *web service*. Um *web service* é uma API que é consumida através da *WWW* (*World Wide Web*). 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. (OLIVEIRA, 2018)

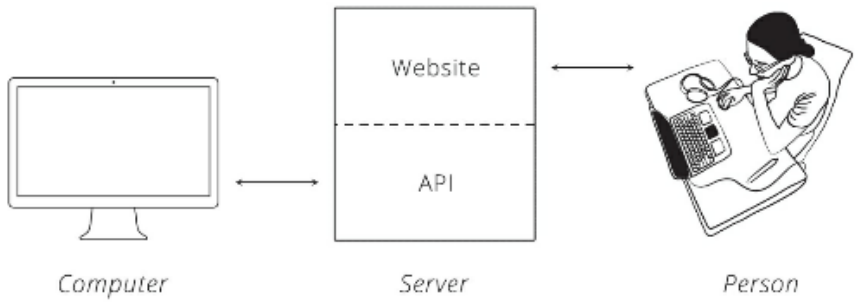
Ao fazer uso do *REST*, tudo é definido em forma de recursos. A disponibilização desses recursos ocorrem através de *URI*s (*Universal Resource Identifier*). Como o próprio nome diz, uma *URI* é utilizada para identificar um determinado recurso.

As APIs (*Application Programming Interfaces*) compõem uma grande parte da web. Em 2013, mais de 10.000 APIs já haviam sido publicadas por empresas para próprio consumo. Isso representa quatro vezes o número de APIs disponíveis em 2010. (COOKSEY, 2014)

Uma API provê um conjunto de funcionalidades que podem ser consumidas por outras aplicações que focam no consumo do serviço e não na implementação do software. O processo de conexão à API é chamado de integração.

A API é uma ferramenta que torna os dados de um website consumível para o computador. É através dela que o computador pode visulizar e editar dados, exatamente como uma pessoa pode carregar páginas e submeter formulários. (COOKSEY, 2014)

Figura 1 – Comunicação com um servidor.



Fonte: (COOKSEY, 2014).

> Quando dois sistemas (*websites, desktops, smartphones*) se conectam através de uma API, nós dizemos que eles estão integrados. Numa integração, você tem dois lados, cada um com um nome especial. Em um lado nós temos o servidor. Este é o lado que, de fato, provê a API. Vale lembrar que, a API é simplesmente outro programa sendo executado no lado do servidor. Ela pode fazer parte do mesmo programa que cuida do tráfego da web ou, pode estar completamente separado. Em ambos os casos, ela está lá, esperando que outros realizem requisições de dados.
>
> O outro lado é o cliente. Este é um programa separado que sabe quais dados estão disponíveis através da API e é capaz de manipulá-los, tipicamente, quando houver uma requisição do usuário. Um ótimo exemplo é um aplicativo de *smartphone* que sincroniza com um website. Quando você aperta o botão de atualizar no aplicativo, ele realiza uma comunicação com o servidor através de API e obtém as novas informações.
>
> O mesmo princípio se aplica aos *websites* que estão integrados. Quando um site envia informações para outro, o site que dispõe da informação está atuando como um servidor e, o site que está recebendo os dados é o cliente. (COOKSEY, 2014)

Diante desse contexto, o presente trabalho tem como objetivo, desenvolver e documentar uma API para realizar o armazenamento de dados georreferenciados dentro de um banco de dados orientado a grafos, permitindo assim, que outras aplicações possam consumir e realizar a aplicação de algoritmos para obter soluções para diversos problemas. A definição da API tem como finalidade uniformizar as principais características existentes entre os *SIG*s (*Sistema de Informação Geográfica*), facilitando a criação e manipulação de dados correlacionados baseados em georreferenciamento.

A definição da API engloba um conjunto de tecnologias, o *ArangoDB*, um banco de dados orientados a grafos, o *QGIS* e o *ArcGIS* que, por sua vez, são sistemas de informação geográfica que permitem analisar, editar e visualizar dados georreferenciados, além de utilizar o *GeoJSON* que é um formato projetado para representar uma variedade de características e estruturas de dados espaciais.

Os objetivos específicos constituem-se em:

- Definir a arquitetura da API REST (*Application Programming Interface*);
- Obter dados geográficos em formato *GeoJSON* a partir de um *GIS* (*Geographic Information System*);
- Converter os dados obtidos para o esquema definido na API REST;
- Gerar os grafos aonde os algoritmos poderão ser aplicados.

O presente documento está organizado da seguinte maneira: o capítulo dois aborda os conceitos fundamentais, necessários para apoiar o estudo apresentado neste trabalho e, além disso, também apresenta as tecnologias empregadas nesse processo, no capítulo 3 é exposto a visão geral da API REST e sua implementação, o capítulo 4 apresenta a prova de conceito e os resultados da API, o capítulo 5 apresenta as considerações finais deste trabalho, além disso, o anexo A exibe todos os recursos e rotas disponíveis na API.

2 REFERENCIAL TEÓRICO

2.1 Web Services

De acordo com o *W3C* (*World Wide Web Consortium*), um *web service* é um sistema de software projetado para suportar interações de máquina para máquina em uma rede. (BOOTH et al., 2004)

A necessidade de serviços interoperáveis entre aplicações vem se tornando cada vez mais comum. A necessidade de autonomia em relação a outros serviços exige que uma estrutura padrão de comunicação seja desenvolvida. Nos dias atuais, aplicações precisam se comunicar com outras para obter o acesso a informação que deseja e também para permitir o desenvolvimento de aplicações descentralizadas e distribuídas. A capacidade de duas aplicações comunicarem-se através de máquinas diferentes, sendo executadas em sistemas operacionais distintos e de atuar em conjunto, é chamada de interoperabilidade. (OLIVEIRA, 2018)

Os *web services* funcionam com a utilização do protocolo da camada de aplicação HTTP. Os clientes que desejam consumir os *web services* precisam, apenas, enviar requisições para os servidores aonde encontram-se os recursos a serem manuseados. (OLIVEIRA, 2018)

Segundo Oliveira (2018), o crescente uso de APIs permitiu que as tecnologias baseadas em *web services* sejam consideradas um requisito para alcançar a escalabilidade e distribuição de seus serviços.

2.1.1 Protocolo HTTP

O *HTTP* (*Hypertext Transfer Procol*) é a base sobre à 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. (OLIVEIRA, 2018)

Para criar aplicações que sigam a arquitetura *REST*, faz-se necessário uma implementação adequada dos verbos *HTTP*, utilizando corretamente suas funções. Para executar uma requisição *HTTP* deve-se levar em consideração três pilares:

- *URI* (*Universal Resource Identifier*) ou *URL* (*Universal Resource Locator*)
- Corpo da requisição
- Verbo da requisição

Entendendo melhor tais pilares, a *URI* ou a *URL* são representadas por um endereço para aonde a requisição será direcionada. Juntamente com a requisição, haverá um corpo

que, mantém informações sobre o *status, header* e até mesmos dados enviados pelo próprio usuário. Para que a requisição possa ser realizada com sucesso, também é preciso utilizar um verbo *HTTP* que será, posteriormente, resolvido pela aplicação e, então, alguma ação deverá ser executada. Os principais verbos *HTTP* são: *GET, DELETE, POST & PUT*.

Uma das principais vantagens de se utilizar os verbos *HTTP* é que, ao desenvolver a aplicação, economiza-se linhas de código, deixando a aplicação mais legível, já que podemos realizar o reaproveitamento de rotas, pois ao chamar uma *URI* com verbos diferentes, ações distintas serão executadas.

2.1.1.1 GET

Utilizado para solicitar uma representação de um recurso, ou seja, um dado. Requisições que utilizem o método *GET* devem apenas retornar dados, não devendo ser utilizado para executar ações.

2.1.1.2 DELETE

Com um nome bastante sugestivo, o método *DELETE* serve para remover um determinado recurso. Ao fazer o uso do verbo *DELETE*, o status de erro 204 deve ser retornado caso o recurso - no qual a ação deve ser executada - seja inexistente.

2.1.1.3 POST

O método *POST* é usado para submeter uma nova entidade a um recurso específico, às vezes causando um mudança de estado no recurso.

2.1.1.4 PUT

Submete uma entidade a um determinado recurso na URI especificada. Caso a requisição esteja se referindo a um recurso já existente, a entidade submetida deve ser considerada como uma modificação àquela já existente no servidor. Caso o recurso não exista, deve ser criado um novo recurso para esta entidade.

2.1.2 REST

Inicialmente apresentado em um capítulo da tese de doutorado de Roy Fielding, o *Representational State Transfer* (*REST*) é um estilo arquitetural, baseado no *HTTP*, utilizado para projetar arquiteturas de software que visam facilitar a interoperabilidade. Segundo Fielding (2000), o *REST* é derivado de outros estilos arquiteturais baseados em rede, combinadas com novas restrições que definem uma nova interface.

O estilo arquitetural *REST* é um abstração dos elementos arquiteturais com sistemas distribuídos e hipermídias. O *REST* foca nas funcionalidades dos componentes, nas restrições

de suas interações com outros componentes, e sua interpretação de elementos de dados significantes. (FIELDING, 2000)

2.1.2.1 Constraints

Por ser derivado de outras arquiteturas, o estilo arquitetural *REST* introduz seis novas características que são chamadas de *restrições REST*. As restrições (*constraints*) *REST* e suas respectivas características são:

1. Cliente-Servidor: a primeira e mais básica restrição do estilo *REST* tem como princípio a separação de responsabilidades. Separando a interface de usuário das responsabilidades do banco de dados, a portabilidade da interface de usuário através de múltiplas plataformas é melhorada, além de aprimorar a escalabilidade simplificando os componentes do servidor. Talvez o mais significativo para a *Web*, no entanto, é que a separação das responsabilidades permite que os componentes evoluam de forma independente. (FIELDING, 2000)

2. *Stateless*: a próxima restrição diz que a comunicação entre cliente e servidor deve ser *stateless*. Isso significa que cada requisição do cliente para o servidor deve conter todas as informações necessárias para se entender a requisição, e não pode obter nenhuma vantagem de qualquer contexto no servidor. Consequentemente, o estado da sessão é mantido no lado do cliente (FIELDING, 2000). Por este motivo, há um maior número de requisições enviadas ao servidor mas, por outro lado, o servidor exige que haja uma implementação correta no lado do cliente.

3. *Cache*: a restrição de cache requer que os dados de uma resposta sejam rotulados, implicitamente ou explicitamente, como informações que podem ou não ficar em *cache*. Isso ocorre devido ao aumento no número de requisições enviadas ao servidor, quando uma resposta enviada pelo servidor pode ficar em *cache*, o cliente recebe o direito de reutilizar aqueles dados posteriormente em requisições equivalente. (FIELDING, 2000)

4. Interface Uniforme: essa restrição é a principal característica do REST e o difere de outros estilos arquiteturais. A interface uniforme está relacionada a recursos, identificadores e representações. Um recurso é qualquer conceito importante no domínio do sistema que desejamos tornar acessível por meio de uma interface uniforme. Um consumidor de serviço endereça recursos por meio de um URI e uma representação. A representação de um recurso refere-se a um documento hipermídia que traz informações úteis sobre o estado do recurso e vincula a outros recursos associados. Finalmente, a representação de um recurso deve ser acessada por uma interface simples que define uma identificação para o recurso e métodos comuns para acessar. Essa interface é a interface uniforme. A restrição de interface uniforme impacta positivamente nos atributos de qualidade de interoperabilidade e de descoberta. (COSTA et al., 2014)

5. Sistema em Camadas: a utilização de sistemas em camadas para separar diferentes funcionalidades tem o intuito de aperfeiçoar o requisito de escalabilidade. A principal desvantagem deste modelo está na adição de overhead e latência nos dados processados, o que resulta em uma queda de performance. Porém, sistemas que façam o uso do *cache* conseguem amenizar essa desvantagem. (DAL; DORNELES; REBONATTO, 2009)

6. *Code-On-Demand*: essa restrição, proposta pelo estilo REST, é opcional. O *Code-On-Demand* permite que clientes possam baixar e executar diretamente código no lado do cliente. Isso permite a simplificação da parte do cliente mas, como *trade-off*, reduz a visibilidade. (COSTA et al., 2014)

2.2 Big Data e NoSQL

A persistência de dados é um detalhe de suma importância e o crescimento das aplicações faz com que o tamanho de dados gerado por elas cresçam rapidamente. Imagine quais seriam os requerimentos necessários para armazenar dados de uma cidade inteira, além disso, o volume de dados cresceria de acordo com o surgimento de novas necessidades. Armazenar tudo isso em um banco de dados relacional geraria um custo extremamente elevado. Além disso, ainda nos depararíamos com as seguintes dificuldades:

- O surgir de novas necessidades exigem mudanças no esquema do banco de dados;
- As mudanças no esquema do banco de dados resultam em sobrecarga de trabalho;
- A gestão solicita que a aplicação atualizada seja lançada o quanto antes.

Segundo Paniz (2016), baseado nessas novas necessidades e novos modelos de aplicações, um grupo de pessoas começou a ressaltar a importância na forma de armazenar os dados. Essas deficiências presentes no RDBMS (*Relational Database Management System*) ou, em português, SGBDR (Sistema Gerenciador de Banco de Dados Relacional), deram origem ao NoSQL (*Not Only SQL*).

Portanto, bancos de dados são considerados como NoSQL quando as seguintes característica se aplicam: capacidade de operar com esquemas flexíveis ou inexistentes, possibilidade de serem distribuídos, API simplificada, não relacional, horizontalmente escalável, suporte nativo a replicação e, eventualmente, consistente, cumprindo as propriedades BASE (*Basic Availability, Soft-state, Eventual Consistency*) e não as propriedades ACID (*Atomic, Consistent, Isolated, Durable*). (NOSQL, 2009)

A SQL (Structured Query Language), ou linguagem de consulta estruturada, é a linguagem de consulta padrão em bancos de dados relacionais. O NoSQL não faz o uso da

SQL e, portanto, são incapazes de compreender consultas elaboradas em SQL. Comumente, bancos de dados em NoSQL possuem sua própria linguagem de consulta, não existindo, necessariamente, um padrão entre elas. Minuciosamente, não recorrer a uma linguagem de consulta estruturada fazem dos bancos de dados NoSQL ferramentas mais poderodas, uma vez que, cada banco tem a possibilidade de implementar peculiaridades que elevem suas funcionalidades.

> Geralmente bancos de dados NoSQL são mais rápidos do que bancos relacionais devido a ausência completa ou quase total do esquema que define a estrutura dos dados modelados. Esta ausência de esquema simplifica tanto a escalabilidade quanto contribui para um maior aumento da disponibilidade. (SILVA, 2017)

Já o termo *Big Data*, segundo Clement (2012), sugere que, conjuntos de dados muito extensos em tamanho são, geralmente, chamados de *Big Data*, sendo esse termo amplamente utilizado para nomear conjuntos de dados muito extensos ou complexos. Os aplicativos de processamentos de dados mais tradicionais não são capazes de trabalhar com esses tipos de dados. As principais características do *Big Data* são:

- Volume
- Velocidade
- Variedade
- Veracidade
- Valor

As duas primeiras características indicam que os dados crescem em grande volume e velocidade. A veracidade se refere ao fato de que os dados devem ser autênticos e estes devem fazer sentido. Segundo Clement (2012), as características variedade e valor são únicas do *big data*.

> Variedade significa que os grandes conjuntos de dados podem ser de diferentes fontes. Isso torna difícil para esses conjuntos de dados serem armazenados em bancos de dados relacionais tradicionais. Embora grandes dados não tenham uma estrutura definida, eles precisam ser tratados de forma diferente.
>
> O valor significa que, de todos os grandes dados que são gerados a partir dessas várias fontes, apenas um pequena parte desses dados tem valor suficiente para gerar decisões empresariais. Ou seja, uma informação em *big data* não é valiosa por conta própria, mas torna-se valiosa no agregado. (CLEMENT, 2012)

Segundo Silva (2017), o NoSQL tem sido bastante explorado quando o assunto é *Big Data*, devido sua capacidade de processar e armazenar grandes volumes de dados, semiestruturados ou não estruturados, que precisam de alta disponibilidade e escalabilidade.

2.2.1 ACID Versus BASE

Em bancos de dados NoSQL, os modelos de consistência de dados às vezes podem ser totalmente diferentes daqueles usados em bancos de dados relacionais. Segundo Sasaki (2015), os dois modelos de consistência mais comuns são conhecidos pelas siglas ACID e BASE. Ambos os modelos possuem suas vantagens e desvantagens.

A seguir, abordaremos as característica de cada um desses modelos.

2.2.1.1 O Modelo de Consistência ACID

Segundo Sasaki (2015), as propriedades ACID garantem que, uma vez concluída a transação, seus dados são consistentes e estáveis no disco. A maioria dos bancos orientados a grafos fazem o uso do modelo de consistência ACID. O ArangoDB utiliza as propriedades ACID.

As garantias que o modelo ACID oferece são:

- Atomicidade (*Atomic*): em uma transação, todas as operação devem ser concluídas com sucesso, caso contrário, cada operação será revertida ao estado anterior;
- Consistência (*Consistent*): ao final de uma transação, o banco de dados deve sair de um estado consistente a um outro estado consistente, mantendo sua integridade;
- Isolamento (*Isolated*): as transação não podem interferir umas nas outras. O acesso aos dados pode ocorrer simultaneamente, mas o resultado das transações ocorre como se elas fossem executadas sequencialmente;
- Durabilidade (*Durable*): os resultados das transações devem persistir, mesmo quando ocorrerem falhas. Uma vez completada a transação, os dados devem estar disponíveis permanentemente.

2.2.1.2 O Modelo de Consistência BASE

Para muitos casos de uso, as transações ACID são extremamente preocupadas com a segurança dos dados, talvez até muito mais do que o necessário. No mundo do NoSQL, as transações ACID não é tão comum, já que alguns do principais bancos de dados alteraram seus requisitos para o modelo BASE com o intuito de obter outros benefícios, como por exemplo, escalabilidade e resiliência. (SASAKI, 2015)

Sua características são:

- Disponibilidade Básica (*Basic Availability*): o banco de dados deve ester disponível na maior parte do tempo;

- Estado Leve (*Soft-state*): os dados e suas réplicas não precisam ser consistentes o tempo todo;
- Eventualmente Consistente (*Eventual consistency*): os dados irão mostrar-se consistentes em algum momento no futuro.

As propriedades BASE valorizam a disponibilidade da aplicação. Por não possuir propriedades severas de persistência dos dados, esse modelo é bem menos consistente do que o modelo que utiliza as propriedades ACID.

2.2.1.3 O Modelo de Consistência do ArangoDB

O ArangoDB (banco de dados escolhido para a aplicação do modelo orientado a serviços georreferenciados) - falaremos, posteriormente, em detalhes sobre ele ainda neste capítulo - provê um ambiente de confiabilidade e consistência em seus dados, fornecendo um ecossistema ideal para se trabalhar com grafos. Obviamente, toda sua durabilidade e persistência de dados ocorre graças ao uso do modelo de consistência ACID.

2.2.2 Classificação dos Bancos de Dados Não Relacionais

Segundo Paniz (2016), existem diversos bancos não relacionais e, normalmente, eles são rotulados de acordo com a forma como os dados são armazenados. Os bancos de dados em NoSQL utilizam um dos seguintes modelos de dados:

- Chave-valor;
- Colunar;
- Documento;
- Grafo.

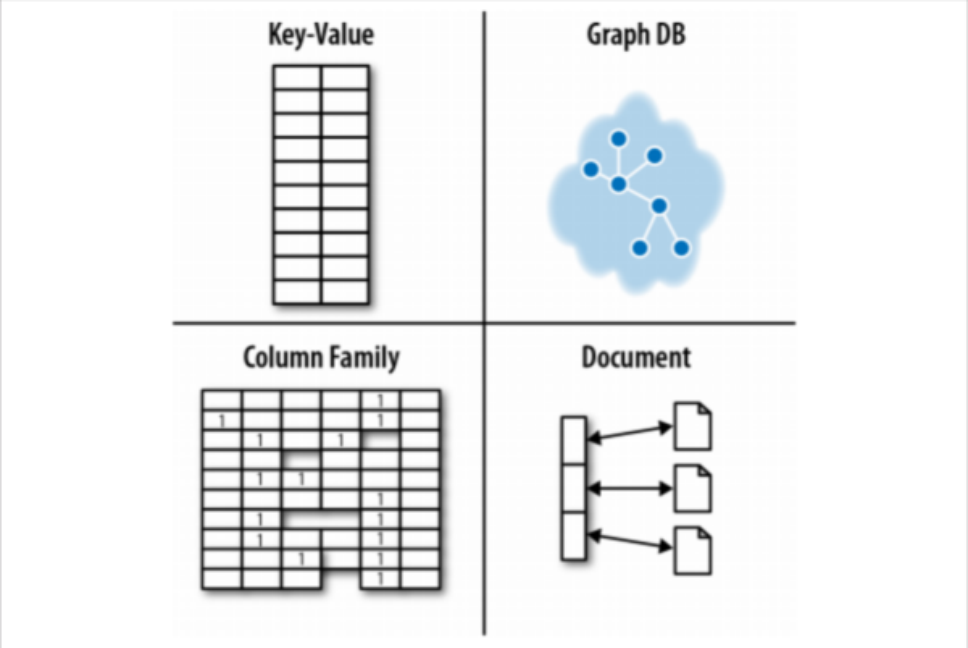
> Cada um deles tem suas vantagens e desvantagens, principalmente na hora de consultar e recuperar os registros. Porém, mesmo dentro do mesmo tipo de banco, existem diferentes implementações que podem ter diferenças de performance e escalabilidade. (PANIZ, 2016)

A Figura 2 ilustra todos estes modelos de bancos de dados não relacionais.

2.2.2.1 Chave-Valor

Esse é, sem dúvidas, o modelo mais simples para dados não estruturados. Ele é altamente eficiente e flexível. Cada registro guarda uma chave única que os identifica, sendo este o único atributo em comum. A desvantagem desse modelo é que seus dados são incapazes de serem autodescritivos.

Figura 2 – Tipos de bancos de dados não relacionais.



Fonte: (EIFREM; ROBINSON; WEBBER, 2015).

2.2.2.2 Colunar

Ideal para conjuntos de dados dispersos, sub colunas agrupadas e colunas agregadas, nele todos os registros fazem parte de uma mesma tabela, porém, cada um de seus registros pode estar em colunas separadas. Segundo Silva (2017), este modelo é adequado para aplicações de mineração de dados (*data mining*) no qual o método de armazenamento é ideal para as operações realizadas nos dados.

2.2.2.3 Documento

Nesse modelo, os registros ficam armazenados em coleções específicas. É um modelo ideal para o armazenamento de repositórios XML (eXtended Markup Language), JSON (JavaScript Object Notation), mapas, listas, objetos autodescritivos e etc.. Mesmo os registros sendo armazenados em coleções específicas, ainda assim, não existe um esquema específico. Como nos bancos baseados em chave-valor, este modelo também atribui uma chave única a cada um de seus registros, mas ao invés de armazenar apenas valores, este modelo armazena objetos inteiros.

2.2.2.4 Grafo

Um banco de dados orientado a grafos é um SGBD (Sistema de Gerenciamento de Banco de Dados), online, com a implementação das operações CRUD (Create, Read, Update, Delete) que expõem um modelo de dados orientado a grafos. Eles são normalmente otimizados para o desempenho transacional e projetados com integridade transacional e

disponibilidade operacional em mente. (EIFREM; ROBINSON; WEBBER, 2015). Um modelo relativamente novo, mas que se mostra altamente eficiente para armazenar grafos com seus nós e relacionamentos.

De acordo com Silva (2017), bancos de dados orientados a grafos podem ser usados em uma vasta gama de aplicações como redes sociais, bioinformática, segurança, controle de acesso, gerenciamento de nuvem e etc.

Em geral, um modelo de banco de dados orientado a grafos é um modelo em que as estruturas de dados para os esquemas e/ou instâncias são modeladas como um grafo direcionado (possivelmente rotulado), ou generalizações da estrutura de dados do grafo, onde a manipulação de dados é expressa por operações orientadas a grafos e construtores de tipos, e restrições de integridade adequadas podem ser definidas sobre a estrutura do grafo. (ANGLES; GUTIERREZ, 2008)

Segundo Angles e Gutierrez (2008), os modelos de bancos de dados orientados a grafos são aplicados em áreas onde a informação sobre o relacionamentos dos dados ou sua topologia é mais importante, ou tão importante quanto o dado em si. Nessas aplicações, os dados e as relações entre eles, geralmente, estão em um mesmo nível. Além de permitir uma modelagem de dados mais natural, e consultas capazes de apontar diretamente para a estrutura do grafo, a implementação de um banco de dados orientado a grafos fornece uma estrutura de armazenamento eficiente, onde algoritmos de grafos possam realizar operações específicas. Como exemplo teríamos algoritmos de custo mínimo e caminho mais longo, subconjuntos de um grafo, inspeção de rotas e etc.

2.3 ArangoDB

O ArangoDB é um banco de dados nativo multi-modelo NoSQL, pois permite o armazenamentos de dados com três dos quatro modelos de dados conhecidos: o modelo chave-valor, o modelo no qual os dados são armazenados como documentos e o modelo de dados orientados a grafos. O ArangoDB também permite que diferentes modelos de dados sejam manuseados em uma mesma consulta. Essa abordagem permite criar aplicações que apresentam alta performance e escalabilidade com os três modelos de dados.

> Os documentos no ArangoDB podem conter um ou mais atributos, cada atributo tem um valor. Um valor pode assumir os seguintes tipos: número, string, booleano, null (nulo), array (vetor) e objeto. Os arrays e objetos podem conter todos esses tipos, o que significa que estruturas de dados arbitrariamente aninhadas podem ser representadas em um único documento.
>
> Estes documentos são agrupados em coleções. Uma coleção pode conter um ou mais documentos. Comparando com os bancos relacionais, as coleções se assemelham com as tabelas e os documentos com as tuplas (linhas). A diferença é que os bancos relacionais possuem colunas que definem um esquema de dados que deve ser seguido por todas as tuplas, no ArangoDB é possível que cada documento detenha seu próprio esquema. (SILVA, 2017)

Segundo Silva (2017), no ArangoDB existem dois tipos de coleções:

- Coleção de documentos: representa uma coleção de vértices no contexto de grafos;
- Coleção de arestas: também são documentos, mas incluem dois atributos especiais, *_from* (que representa a origem) e *_to* (que representa o destino). Esses atributos representam as relações entre os documentos.

Geralmente, dois documentos (vértices) são armazenados em uma coleção de documentos que, por sua vez, são relacionadas por um outro documento (aresta) armazenado dentro de uma coleção de arestas, respectivamente. As coleções são armazenadas dentro de uma base de dados, sendo possível haver uma ou mais bases de dados. As consultas no ArangoDB são escritas usando a linguagem AQL (ArangoDB Query Language). (SILVA, 2017)

2.3.1 Vantagens do ArangoDB

De acordo com ArangoDB (2017), suas principais vantagens são:

- Consolidação
- Escala de Desempenho Simplificado
- Complexidade Operacional Reduzida
- Forte Consistência de Dados
- Tolerância a Falhas
- Custos Minimizados
- Transações

2.3.1.1 Consolidação

Como um modelo de banco de dados multi-modelo nativo, o ArangoDB minimiza os componentes que você precisa mantar, reduzindo a complexidade da pilha de tecnologia para sua aplicação ou uso. Isso significa que haverá um menor custo total, aumentando a flexibilidade e consolidando suas necessidades técnicas. (ARANGODB, 2017)

2.3.1.2 Escala de Desempenho Simplificado

Com o crescimento e amadurecimento das aplicações ao longo do tempo, o ArangoDB permite reagir facilmente ao crescimento do desempenho e das necessidade de armazenamento, escalando independentemente com diferentes modelos de dados. O ArangoDB escala tanto na vertical quanto na horizontal, e se o seu desempenho precisar diminuir, você pode,

facilmente, reduzir a escala do seu sistema de *back-end* para economizar em hardware e requisitos operacionais. (ARANGODB, 2017)

2.3.1.3 Complexidade Operacional Reduzida

Certas tarefas requerem um banco de dados que utiliza o modelo de documentos, enquanto outros exigem um banco de dados de grafos. Um banco de dados nativo multi-modelo permite a você utilizar diferentes modelos sem complexidade, mas com a consistência de dados de um sistema tolerante a falhas. O ArangoDB permite a você escolher o modelo de dados certo para o trabalho certo. (ARANGODB, 2017)

2.3.1.4 Forte Consistência de Dados

Ao usar múltiplos bancos de dados de modelo único, a consistência dos dados torna-se um problema. Esses bancos de dados não devem falar um com o outro, significando que você precisa implementar alguma forma de funcionalidade de transação para manter seus dados consistentes entre diferentes modelos. Com o ArangoDB, um único back-end gerencia seus diferentes modelos de dados com suporte para transições ACID, assim, o ArangoDB fornece consistência forte. (ARANGODB, 2017)

2.3.1.5 Tolerância a Falhas

Construir sistemas tolerantes a falhas com muitos componentes não relacionais é uma tarefa desafiadora em si. Ao trabalhar com clusters, essa dificuldade cresce ainda mais. A implantação e a manutenção desses sistemas requerem conhecimentos profundos de várias tecnologias e pilhas de tecnologia. Além disso, reunir vários subsistemas que foram projetados para executar independentemente impõe custos significativos de engenharia e operação.

A solução para esses desafios é um banco de dados multi-modelo e uma pilha de tecnologia consolidada. Por design, o ArangoDB permite arquiteturas modernas e modulares com diferentes modelos de dados em execução e também funciona para o uso de cluster. (ARANGODB, 2017)

2.3.1.6 Custos Minimizados

Cada tecnologia de banco de dados que você usa precisa de manutenção contínua, patches, correções de erros e outras modificações entregues pelo fornecedor. Cada nova atualização deve ser testada por um membro da equipe especializada e testada quanto ao ajuste geral no sistema atual. O uso de um banco de dado multi-modelo reduz significativamente esses custos de manutenção, pois permite reduzir o número de tecnologias de banco de dados que você precisa para sua aplicação. (ARANGODB, 2017)

2.3.1.7 Transações

É um verdadeiro desafio fornecer garantias transacionais em várias máquinas e apenas algumas bases de dados NoSQL fornecem essas garantias. Como um banco de dados nativo multi-modelo, o ArangoDB requer transações para garantir a consistência dos dados. O ArangoDB fornece consistência forte quando executado no modo de cluster. (ARANGODB, 2017)

2.4 Teoria dos Grafos

Um grafo consiste em um conjunto finito de vértices e arestas. Também pode ser descrito como um conjunto de nós conectados por arcos, também podendo ser chamado de relacionamentos.

Na teoria dos grafos, a definição de um grafo é dada por $G = (V, E)$, onde $V = (v_1, v_2, ..., v_n)$ representa um conjunto de vértices e $E = (e_1, e_2, ..., e_n)$ representa um conjuntos de arestas.

De modo geral, representamos objetos como vértices ou nós, e representamos uma relação entre dois objetos como uma aresta, também conhecido como arco, que conecta seus vértices. (STEIN; DRYSDALE; BOGART, 2013)

Os grafos estudam as relações entre os objetos de um determinado conjunto, um dos seus exemplos mais famosos é o do caixeiro viajante, que busca em um grafo o seu caminho mínimo. O problema do caminho mínimo consiste em buscar o menor caminho entre um par de vértices em um grafo. Este é um problema fundamental com inúmeras aplicações. Uma área comum para a aplicação do problema do caminho mínimo é a de logística, onde, produtos que se encontram em um determinado local precisam ser conduzidos para outro lugar com o menor custo possível. (SILVA, 2017)

> O problema do caixeiro-viajante é um problema de caminho ponderado mínimo com fortes restrições sobre a natureza do caminho, de modo que este caminho pode vir a não existir. No problema de caminho mínimo, não colocamos restrições (além da referente ao peso mínimo) na natureza do caminho; como o grafo é conexo, sabemos que o caminho existe. Por esta razão, podemos esperar que haja um algoritmo eficiente para resolver este problema, ainda que não exista algoritmo eficiente para o problema do caixeiro viajante. (GERSTING, 2013)

Quando o assunto é caminho mínimo, o algoritmo de Dijkstra é um dos mais conhecidos na área, sendo ele ideal para resolver problemas em que é preciso deslocar-se de um local para outro, apontando qual trajetória oferece o menor caminho.

> Outra forma de resolver o problema do caixeiro é por meio da utilização de métodos heurísticos. Uma heurística é uma técnica algorítmica para encontrar

> soluções para problemas de otimização em tempo hábil, no entanto, sem oferecer garantias quanto à sua qualidade em relação às soluções ótimas. (SILVA, 2017)

Segundo Sucupira (2007), as heurísticas específicas frequentemente possuem alta qualidade, mas, na quase totalidade dos casos, não são flexíveis o suficiente para contribuírem de maneira significativa na resolução dos problemas de otimização em geral.

Segundo Silva (2017) as hiper-heurísticas surgiram como uma consequência da necessidade de elevar o nível de generalidade dos métodos de otimização. O diferencial das hiper-heurísticas é sua capacidade de resolver uma variedade de problemas, ao invés de limitar-se a uma situação em específico.

Em outras palavras, uma hiper-heurística recebe um conjunto de heurísticas e, em cada ponto de decisão, os métodos de alto nível serão responsáveis por selecionar uma delas para aplica-lá em um dado momento. A aplicação dessas heurísticas irá depender do estado atual do problema ou de um determinado passo. (SILVA, 2017)

As meta-heurísticas se aplicam à problemas em que não há o conhecimento de um algoritmo capaz de resultar numa solução satisfatória ou quando não houver um método ótimo diante de um determinado problema.

2.5 Sistema de Informação Geográfica

Uma das principais utilidades de um SIG (Sistema de Informação Geográfica) é a manipulação de dados geoespaciais no que se refere a informações sobre a localização geográfica de uma entidade. Isto envolve frequentemente a utilização de coordenadas geográficas, como a latitude e longitude. Além de dados espaciais, outros termos também são comumente usados, como por exemplo: dados geoespaciais, dados geográficos, dados de SIG, dados cartográficos, dados de localização, coordenadas e geometria espacial. (CENSIPAM, 2010)

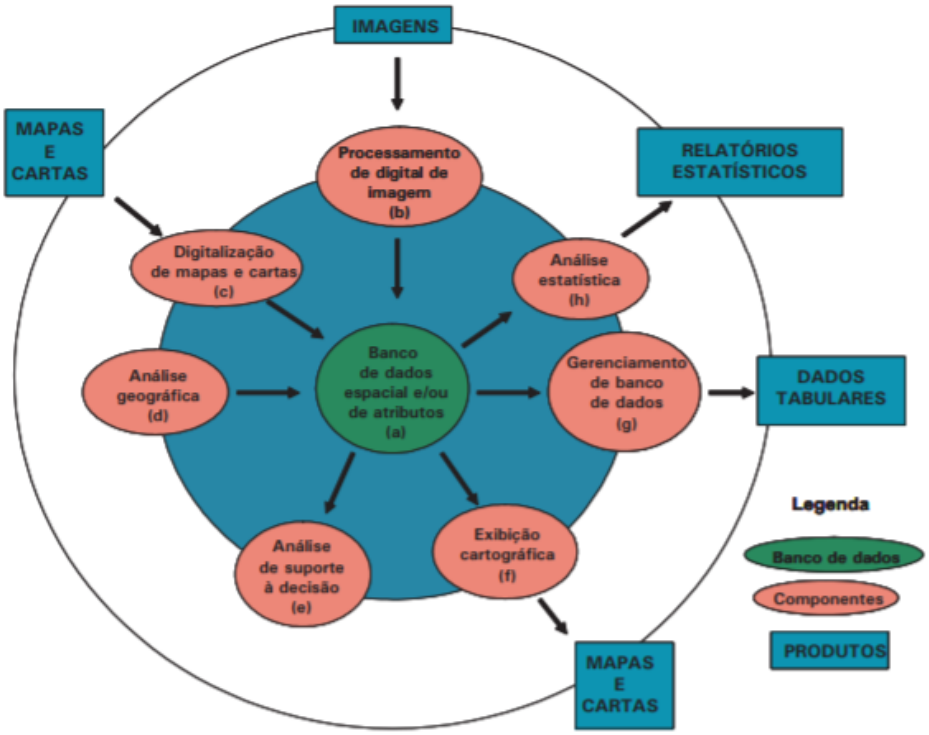
Segundo CENSIPAM (2010), muitas pessoas erroneamente supõem que as aplicações geoespaciais apenas produzem mapas, mas a análise dos dados geoespaciais é outra função primária de aplicações geoespaciais.

Na Figura 3, podemos ver os elementos que, geralmente, compõem um SIG. Note que, nem todos os SIGs apresentam todos os elementos representados na Figura 3, mas os elementos básicos estão presentes em qualquer SIG.

2.5.1 QGIS

O QGIS é um SIG (Sistema de Informação Geográfica) de código aberto. Através do QGIS, temos um poderoso conjunto de ferramentas onde é possível coletar, visualizar, gerir,

Figura 3 – Componentes de um Sistema de Informações Geográficas.



Fonte: (HAMADA, 2007).

editar, analisar dados, criar mapas para impressão, em geral, é uma ferramenta especializada em manipulação de dados espaciais do mundo real.

O QGIS representa um etapa crucial no desenvolvimento deste trabalho, sendo um passo importante durante o estágio de compreensão dos SIGs.

2.5.2 ArcGIS

Também utilizado para criar e visualizar mapas, compilar informação geográfica, análise de informações mapeadas, compartilhamento e descoberta de dados geográficos, o ArcGIS é outra ferramenta de SIG poderosa.

O ArcGis será nossa principal fonte de dados, nele obteremos os dados georreferenciados vindo de fontes como mapas e tabelas de atributos.

2.5.3 GeoJSON

Baseado no JSON (RFC-7159), o GeoJSON (RFC-7956) é uma notação utilizada para representar uma variedade de estruturas de dados geográficos. De acordo com Butler et al. (2016), este formato define uma variedade de objetos JSON e a maneira como eles são combinados para representar dados com características geográficas, suas propriedades,

e suas extensões espaciais. O GeoJSON usa um sistema de referência de coordenadas geográficas, o WGS (World Geodetic System) 1984, e unidades de grais decimais.

Os tipos geométricos suportados pelo GeoJSON são: *Point, LineString, Polygon, MultiPoint, MultilineString, MultiPolygon, e GeometryCollection*. Um objeto em GeoJSON armazena objetos geométrico e propriedades adicionais, já um conjunto de recursos é armazenado dentro de um objeto chamado *FeatureCollection*, ou somente *Feature*.

O formato está preocupado com os dados geográficos no sentido mais amplo. Qualquer coisa com qualidades que estão limitadas no espaço geográfico pode ser um recurso, independente de ser ou não ser uma estrutura física. Os conceitos apresentados pelo GeoJSON não são novos, mas sim derivados a partir de padrões já existentes do SIG, sendo apenas simplificados para facilitar o desenvolvimento de aplicações *Web* utilizando o JSON. (BUTLER et al., 2016)

Figura 4 – Exemplo de um objeto em GeoJSON que representa um ponto nas ilhas Dinagat.

Fonte: (GEOJSON, 2017).

2.6 Node.js

O Node.js é uma plataforma construída em cima da engine V8 do *Chrome*, possuindo a capacidade de interpretar códigos escritos em JavaScript e os transformar em bytecode. O Node.js utiliza um modelo orientado a eventos que o torna leve e eficiente. O grande diferencial, e conceito novo, do Node.js é que ele permite executar JavaScript diretamente no servidor.

O JavaScript é uma linguagem de programação interpretada, multi-plataforma e orientada a objetos, projetada em 1995 por Bredan Eich, e surgiu pela primeira vez nos *browsers*

Netscape. Seu desenvolvimento teve como foco complementar as funcionalidades das páginas *Web* através de código que seriam executados do lado do cliente, diretamente no browser. O propósito era tornar as páginas *Web* mais interativas, apresentando mais interação com o usuário, comunicação assíncrona com outras aplicações *Web*, bem como alterar, de forma dinâmica, o conteúdo apresentado. (SOUSA, 2015)

Para fornecer uma maneira simples e eficiente de criar aplicações de redes escaláveis o Node.js faz o uso da programação orientada a eventos e possui um modelo de entrada e saída de dados não bloqueante. Seu ciclo de vida dispensa o uso de uma nova *thread* para cada operação, ao invés disso, ele é capaz de executar múltiplas operações em background. Quando uma operação é completada, o seu *callback* é adicionado a fila para ser, posteriormente, executado.

> Ao contrário do que acontece com outros ambientes de desenvolvimento, o Node.js não depende de *multithreading* para a execução de processos concorrentes na lógica de negócio do servidor. Na realidade, deve-se entender o Node.js como uma plataforma de desenvolvimento de servidores Web que possibilita a construção de sistemas altamente escaláveis, sem as complexidades de gestão de um sistema *multithreading*, operando apenas em *single-thread*, através de uma abordagem baseada em eventos e de input e output não bloqueante.
>
> Deste modo, esta tecnologia utiliza operações de I/O não bloqueantes, assíncronas ou orientadas a eventos, simplesmente registando uma função de callback que é invocada quando a operação em causa é concluída, prevenindo assim que a aplicação não fique bloqueada aquando da execução deste tipo de operações. Com isto é assim possível mudar o paradigma tradicional, no qual a maioria das operações de I/O são bloqueantes, suspendendo a execução do programa principal até que as operações sejam concluídas, para uma abordagem onde todas as operações podem ser executadas de forma assíncrona, não bloqueante, criando todo um sistema como base num modelo de eventos. (SOUSA, 2015)

Considerado o maior ecossistema de bibliotecas de código aberto do mundo, o Node.js possui um ecossistema de pacotes chamado *NPM* (*Node Package Manager*). O núcleo de uma aplicação em Node.js é composto tanto por pacotes próprios quanto por módulos disponibilizados pela comunidade.

2.7 Express

O Express é um *framework* minimalista e flexível desenvolvido para ser utilizado com o Node.js, dispondo de uma camada de funcionalidades fundamentais para o desenvolvimento de aplicações web sem comprometer o uso do Node.js.

> O *Express.js* é um *framework* que tem por objetivo simplificar o desenvolvimento de APIs para aplicações *web* com *Node.js*, permitindo a criação de servidores que receba requisições *HTTP* de forma simples. Além disso fornece maneiras de reutilizar código e provê uma estrutura semelhante ao modelo

> *MVC* (*Model View Controller*). O *Express.js* abstrai a complexidade do servidor HTTP do *Node.js* adicionando um número significativo de funcionalidades. (SILVA, 2017)

3 A API PROPOSTA

Neste capítulo será apresentado a visão geral da API REST desenvolvida, seus princípios e os passos de como foi feita a implementação dos fundamentos REST.

3.1 Visão Geral

Na atualidade, muitos sistemas de informações geográficas possuem estruturas de dados específicas. Mesmo que embora tais dados possam ser exportados para estruturas homogêneas que seguem alguma notação, seus utilizadores ainda precisam se preocupar com propriedades específicas do SIG utilizado e com o desenvolvimento de uma aplicação para realizar o tratamento de tais informações. O uso de estruturas de dados específicas impede que aplicações de terceiros possam reutilizar tais informações com facilidade. O desenvolvimento de ferramentas que aprimoram as funcionalidades de um SIG pode prolongar-se por muito tempo, além disso, pode tornar-se um trabalho árduo e, por diversas vezes, impraticável.

Além disso, o desenvolvimento de aplicações ou APIs podem limitar-se a uma determinada arquitetura ou ambiente, impossibilitando a interoperabilidade de sistemas e seu uso em diferentes dispositivos. Preocupar-se com o desenvolvimento de inúmeras aplicações para contemplar diversos ecossistemas pode acabar se tornando uma tarefa complexa e de alto custo.

Os dados georreferenciados fornecidos pelos atuais sistemas de informações geográficas são devidamente padronizados de acordo com a especificação do GeoJSON. Porém, os dados fornecidos pelos SIGs não possuem apenas informações geográficas, elas armazenam também características específicas da aplicação de onde foram exportadas.

Para solucionar tais problemas, este trabalho propõe a implementação de um Web Service capaz de lidar com tais informações, uniformizando-as e fornecendo um consumo de operações básicas que possam auxiliar no desenvolvimento de outras aplicações através da plataforma Web. Além da API, vamos definir um modelo, com base nas notações já existentes, que auxilie na integração de informações de diferentes fontes.

O modelo trata-se de uma estrutura de dados flexível que nos permita criar um aglomerado de informações e que possam ser tratadas como um grafo com propriedades particulares. Portanto, é preciso atender os seguintes requisitos:

- Possuir informações básicas de georreferenciamento;
- Dispor do estado original dos dados;
- Estar de acordo com as especificações do GeoJSON.

3.2 Implementação

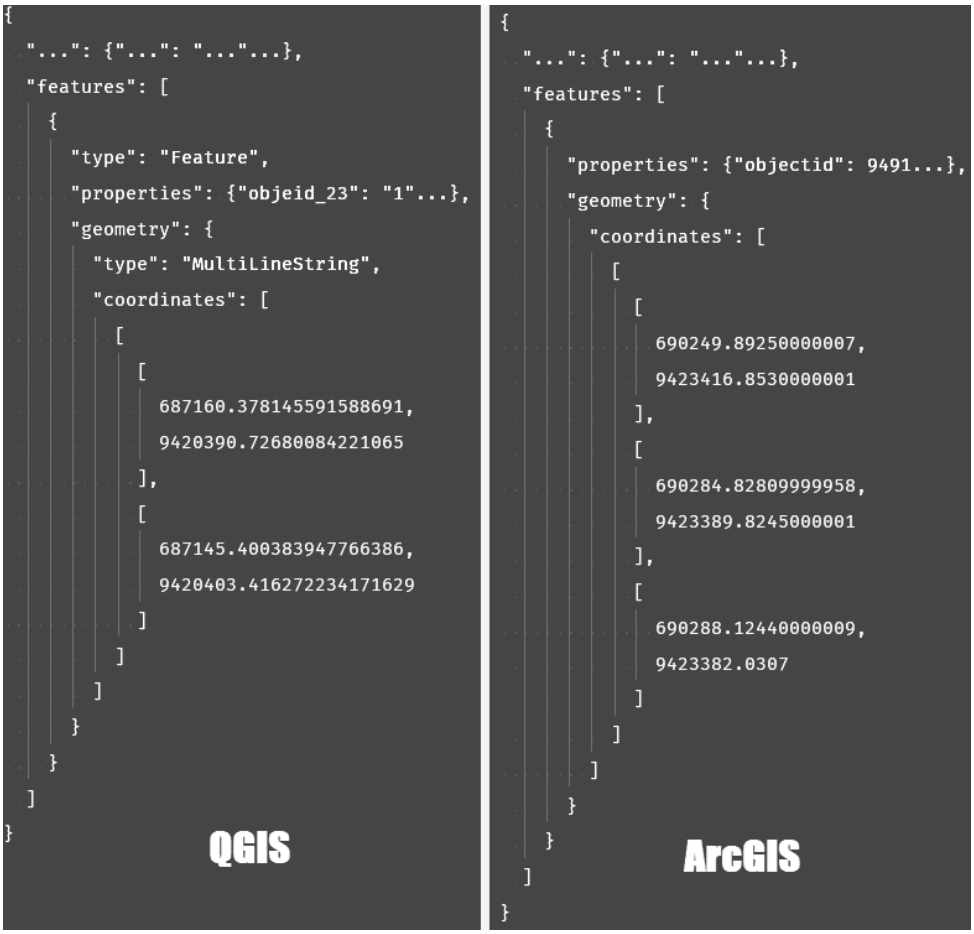
Esta seção aborda todos os requisitos necessários para a implementação da API REST para consumo dos dados georreferenciados.

3.2.1 Output

A base de dados tratada neste trabalho será construída a partir do mapa das ruas da área urbanizável da cidade de Mossoró, Rio Grande do Norte, Brasil. Tais dados guardam informações sobre suas coordenadas geográficas, identificação do trecho de via, sua extensão e etc. Os dados aqui tratados são dados oficiais, cedidos pela Prefeitura do Município de Mossoró.

Para realizar a elaboração do nosso modelo é preciso primeiro compreender os dados que serão tratados em nosso *Web Service*. Iremos analisar o output de dados que foram criados por dois dos principais sistemas de informação geográfica existentes atualmente: ArcGIS e o QGIS.

Figura 5 – Comparação do GeoJSON gerado por diferentes Sistemas de Informações Geográficas.



Fonte: autoria própria.

A estrutura do *output* gerado por ambos os SIGs mostram-se semelhantes, pois, são definidos com base nas especificações do GeoJSON. Apesar de cada SIG possuir atributos particulares, ambos possuem em comum uma coleção de *features*, que é um array de *features*. Uma *feature*u representa uma área espacial limitada e dentre seus diversos atributos, possui um *type*, um objeto *geometry* (aqui estará as coordenadas da área especial em questão), um objeto *properties* contendo informações características de tal domínio e um atributo *id* que será seu identificador.

Independentemente do QGIS e do ArcGIS possuírem atributos peculiares, ambos possuem uma coleção de *features*, é nela que encontra-se a informação relevante para o desenvolvimento deste trabalho. Além disso, cada *feature* conta com o atributo *properties* que inclui dados essenciais para a identificação da região em questão.

Dentro de cada *feature* há um *array* de coordenadas geográficas que, quando traçados, representam um determinado trecho de via. O atributo responsável por armazenar tais informações é designado por *coordinates*. Além disso, é de grande valia as informações guardadas dentro de *properties*, é através deste objeto que poderemos obter informações, tais como, por exemplo, o nome da rua a qual o trecho de via em questão está relacionado.

Será utilizado os *features* inclusos no *output* gerado pelo ArcGIS para popular nossa base de dados, fazendo o uso de alguns atributos do objeto *properties* e das coordenadas geográficas armazenadas no array de *coordinates*. Por isso, é preciso entender quais os dados contidos dentro do objeto *properties* serão necessários para o desenvolvimento deste trabalho. A Figura 6 exibe detalhes dos atributos contidos dentro do objeto *properties*. Dentre as propriedades ali contidas, as que se farão úteis para popular nossa base de dados são: identificador do trecho, identificador do logradouro, classificação da via, o nome do logradouro e o comprimento da via. Tais propriedades são representadas, respectivamente, pelos seguintes atributos: idtrechovia, idlogradouro, classificacaoctb, nomeusual e *st_length(shape)*.

3.2.2 O Modelo

Para começar a explicar o modelo demonstrado no diagrama de classe da **??**, iremos partir do princípio de que as ruas a serem cadastradas são compostas por um conjunto de trechos de vias que, por sua vez, são constituídos por um intervalo de duas coordenadas, c_1 e c_2, indicando o início e o fim, de modo linear, de uma determinada extensão da via. Tais coordenadas c_1 e c_2 são consideradas como pontos de interesse, podendo fazer parte, ou não, de fragmentos de trechos de outras vias.

Uma via será representada pelo *GeoRoute*. Um *GeoRoute* possui uma lista de propriedades (*listProperties*) que contém informações essenciais para sua identificação como por exemplo: nome da rua, classificação e etc. Não menos importante, também há um *array* de coordenadas geográficas (*listGeoPoint*), onde será armazenado os pontos da via em questão e que serão, também, representadas através de um *InterGeoRoute*. Além disso, haverá um

Figura 6 – Conteúdo do objeto *properties*, contido dentro de uma *feature*.


```
"properties": {
  "objectid": 9491,
  "idtrechovia": 9491,
  "idlogradouro": 1780,
  "idhierarquizacaoviaria": 0,
  "nomeusual": "Rua Professora Maria Amelia Gurgel",
  "observacao": " ",
  "idformulario": 0,
  "idzonahomogenea": 0,
  "idfatorvizinhanca": 0,
  "idlogradourosiat": 0,
  "classificacaorelevancia": "Outros",
  "classificacaoctb": "Arruamento",
  "idareaespecialadensamentourbano": null,
  "idareaespecialassentamentorural": null,
  "idareaespecialdireitopreempcao": 1,
  "idareaespecialinteresseindustri": null,
  "idareaespecialinteressesocial": null,
  "idareaespecialpreservacaoambien": null,
  "idareaespecialsegurancaalimenta": null,
  "idareaespecialurbanacentral": null,
  "idareaexpansaourbana": null,
  "idconeaproximacaoaeroporto": null,
  "idzonaurbana": 1,
  "st_length(shape)": 52.632737628180664
},
```


Fonte: autoria própria.

objeto (chamado de *listPropertiesSrc*) que preservará as informações originais da sua fonte de origem. Por fim, existirá um atributo *length*, utilizado para representar o comprimento, em metros, da via.

No *InterGeoRoute* estão contidas as coordenadas c_1 e c_2 de um dado ponto. Sua coordenada geográfica representa o ponto de encontro de duas linhas ou de dois planos que se cortam. Um *InterGeoRoute* pode estar presente em diversos trechos de vias.

Um *IPoint* será um ponto de interesse, ou seja, uma localização específica de grande utilidade ou bastante interessante. Um *IPoint* poderá ser formado por um ou mais *InterGeoRoute* de modo a representar, geograficamente, um ponto, uma linha ou um polígono. Será constituído por um *array* de *listGeoPoint* e um *array* de *listIdRef* que irá reunir os *InterGeoRoute*s referentes à cada ponto geográfico, respectivamente, apresentado no *listGeoPoint*. Além disso, pode possuir um atributo *type* que representará o tipo deste IPoint. Para o desenvolvimento deste trabalho, definiremos o *type* dos nossos *IPoint*s como sendo do tipo *Intersection*.

Para representar um trecho de via, utilizaremos o *AbsGeoRoute*. Nele haverá dois

*IPoint*s, usados representar um ponto inicial e final, retratados pelas propriedades *_from* e *_to*, respectivamente.

3.2.3 Conversão dos Dados

Durante o processo de implementação da API REST, foi preciso desenvolver uma ferramenta de linha de comando para auxiliar na transformação dos dados gerados pelo ArcGIS e, então, convertê-los para o modelo adotado em nosso banco de dados, de modo que os dados venham a ser importados, de uma só vez, para a nossa base através da interface do ArangoDB. Devido a grande massa de dados (totalizando 31.963 registros) e com o intuito de evitar sobrecargas no servidor utilizado, o uso da API para efetuar tais cadastros foi descartado, limitando seu uso, apenas, para realizar a prova de conceito.

Também desenvolvido com o Node.js, esta CLI (*Command Line Interface*) possui quatro diferentes algoritmos que podem ser executadas sob o *output* de dados gerado com base na notação do GeoJSON. Os algoritmos disponíveis e suas respectivas funções são:

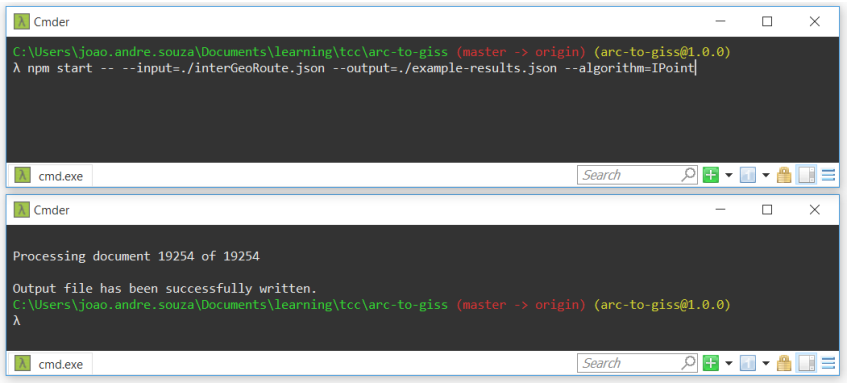
- *AbsGeoRoute*: gera uma lista de AbsGeoRoute. Um AbsGeoRoute é constituído por dois *IPoints*: *_from* & *_to*. Um AbsGeoRoute é uma rota abstrata representada por dois pontos de interseção.
- *GeoRoute*: gera uma lista de *GeoRoute*. Cada item da lista possui um objeto contendo informações sobre um caminho ou via que está relacionado a dois ou mais *InterGeoRoute*. O seu ponto inicial e final devem ser representados por um *InterGeoRoutes*, respectivamente.
- *InterGeoRoute*: gera uma lista de *InterGeoRoute*. Cada item da lista possui um *array* de coordenadas geográficas, denomidado *listGeoPoint*.
- *IPoint*: gera uma lista de *IPoints*. Um *IPoint* pode representar um ponto em específico, uma linha ou um polígono. É constituído por um *listGeoRoute* (um *um array de coordenadas geográficas*), um *listIdRef* (um *array* com os identificadores *InterGeoRoute* respectivos a cada uma das coordenadas dentro do objeto *listGeoRoute*) e um atributo *type* (neste trabalho iremos apenas lidar com interseções, por tal motivo, todos os *IPoints* serão do tipo *Intersection*) que indica o tipo deste IPoint.

O uso da CLI ocorre em conjunto com o NPM. Através do comando *start* do NPM podemos executar a CLI passando três argumentos. São eles:

- *input*: caminho até o arquivo de saída em formato GeoJSON.
- *output*: caminho aonde será salvo o arquivo após executada as operações contidas na CLI.

- textitalgorithm: um dos quatros algoritmos disponíveis a serem executados (*AbsGeoRoute, GeoRoute, InterGeoRoute, IPoints*).

Figura 7 – Exemplo de uso da CLI utilizada para conversão dos dados.



Fonte: autoria própria.

A Figura 7 exemplifica o uso da CLI, no terminal de cima é exibido o comando e os argumentos utilizados, logo após, o terminal exibe os documentos processados e a mensagem de conclusão. Após o manuseio apropriado da CLI podemos obter os dados em formatação adaquada para, então, serem importados pela interface de importação do banco de dados utilizado, o ArangoDB, em sua respectiva coleção ou aresta.

3.2.4 API REST

A implementação da API REST foi realizada através do uso do Node.js e do *framework* Express, utilizando a linguagem JavaScript. O uso do Express simplifica o desenvolvimento de APIs através de sua capacidade de abstração dos recursos do Node.js, além disso, o uso do Express ao invés do FoxFramework do ArangoDB permitirá a API comunicar-se mais facilmente com outros bancos, não limitando-se ao ArangoDB. O banco de dados NoSQL utilizado foi o ArangoDB, pois possui a capacidade de trabalhar com coleções de documentos e estruturas de grafos. Através de suas ferramentas e funcionalidades, o ArangoDB nos permitirá executar buscas em profundidade quando dado um determinado ponto de interesse. A escolha do ArangoDB ocorreu devido ao fato de possuir-se conhecimento e experiências anteriores em tal tecnologia.

3.2.5 Banco de Dados

A API comunica-se com o ArangoDB através do pacote oficial, disponibilizados pelo NPM. O módulo do NPM utilizado será o ArangoJS, que é um middleware, funcionando como uma ponte entre a API e o banco de dados e, fornecerá a comunicação entre ambas as aplicações.

O ArangoDB é um banco multiuso, pois trabalha com mais de um modelo de persistência, o que permitiu uma flexibilidade bem maior para o desenvolvimento deste trabalho. Para a implementação da API foram utilizados dois modelos: o modelo orientado a documentos e o modelo orientado a grafos. Foram criadas quatro coleções no ArangoDB, sendo elas dividas em documentos e orientadas a grafos.

As seguintes coleções são orientadas a grafos:

- AbsGeoRoute
- GeoRoute

Já as coleções abaixo foram criadas utilizando o modelo de documentos:

- InterGeoRoute
- IPoint

Todas as coleções citadas acima foram necessárias para a implementação dos requisitos a serem atendidos pela API desenvolvida. O nome de cada coleção é sugestivo ao conteúdo que armazena e ao seu uso na API.

4 PROVA DE CONCEITO

Este trabalho resultou no desenvolvimento de uma API RESTful, utilizando como base os padrões definidos no estilo arquitetural REST e implementou todos os fundamentos idealizados para realizar o tratamento dos dados georreferenciados quando utilizada a notação do GeoJSON. Para validar a API, foi utilizado um servidor local para a execução da API e, requisições foram executadas com o intuito de obter e legitimar os resultados obtidos.

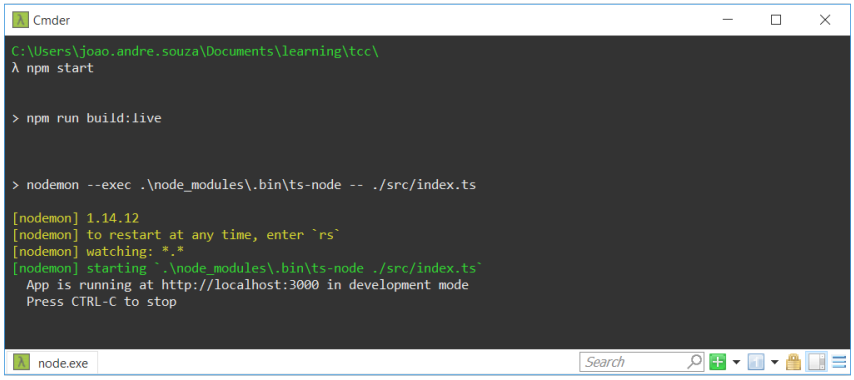
4.1 Banco de Dados

Para a execução do banco de dados, uma instância online do ArangoDB foi disponibilizada pelo LORDI (Laboratório de Redes e Sistemas Distribuídos) da UERN (Universidade do Estado do Rio Grande do Norte). Com o banco de dados online e em funcionamento, foi possível realizar o consumo dos dados contidos nesta instância através do uso da API.

4.2 Implementando a API

Para a disponibilização da API, foi levantada uma instância local. O processo de levantamento da API não exigiu configurações adicionais e pôde comunicar-se normalmente com a instância online do banco de dados. O seu levantamento ocorre através do comando *npm start*.

Figura 8 – Instância da API REST em execução.



Fonte: autoria própria.

A Figura 8 exibe o *console* referente a instância da API REST. Através do Node.js a API é executada em modo de desenvolvimento na porta 3000 (quando em produção será executado na porta 80), e pode ter sua instância reiniciada a qualquer momento com o uso do comando *rs*. A execução em modo de desenvolvimento ocorreu devido ao fator da API REST ser executada em uma máquina local.

4.3 Rotas

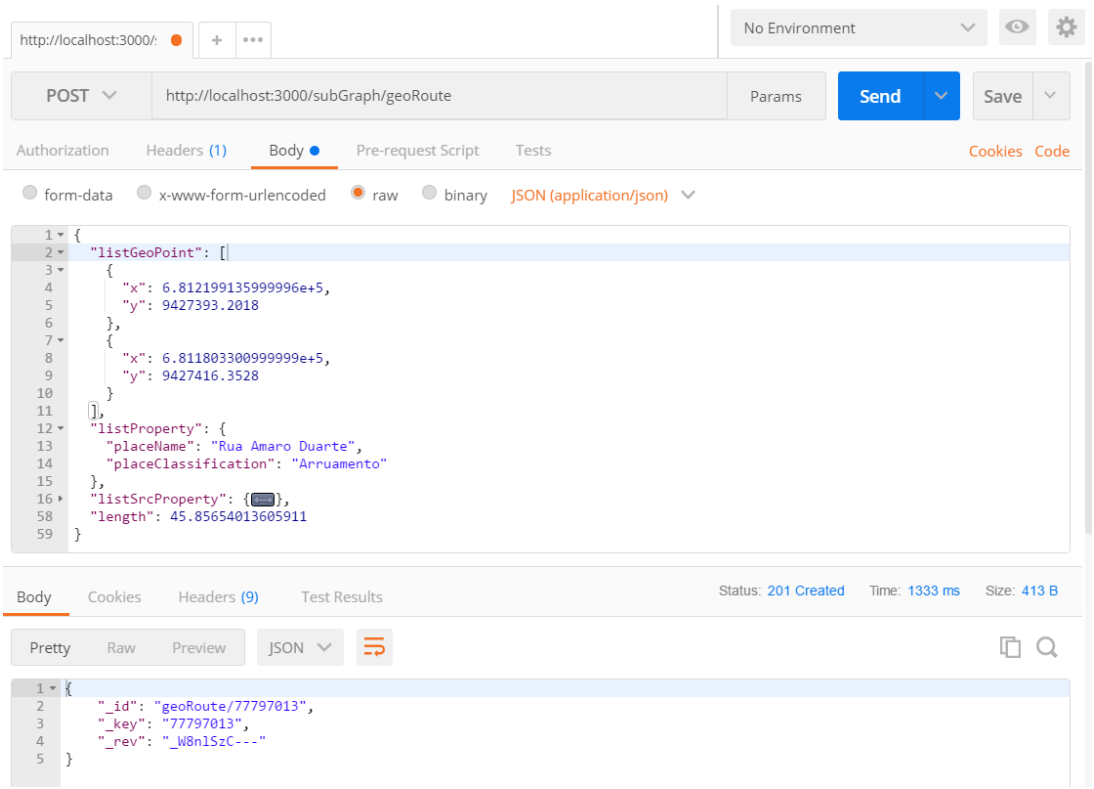
Os recursos disponíveis foram disponibilizados por um URI (*Uniform Resource Identifier*) e é uma das restrições apresentadas na interface uniforme idealizada por Fielding (2000). Para mostrar o funcionamento da API, suas principais rotas serão abordadas nesta seção, exemplificando o seu trabalho. Ao todo são quatro recursos e 26 rotas disponíveis na API.

4.4 Criar GeoRoute

Para a criação de uma via na API, temos que enviar uma requisição, através do método POST, no recurso geoRoute. No corpo da mensagem temos que enviar um objeto JSON contendo os atributos *listGeoPoint, listProperty, listSrcProperty* e *length*.

O *listGeoPoint* contém todas as coordenadas geográficas que constituem a via em questão. No *listProperty* enviaremos os atributos de maior relevância, enquanto que, o *listSrcProperty* manterá o estado original de sua fonte. Por fim, temos o atributo *length*, que contém a extensão da via.

Figura 9 – Resultado da adição de um novo registro.

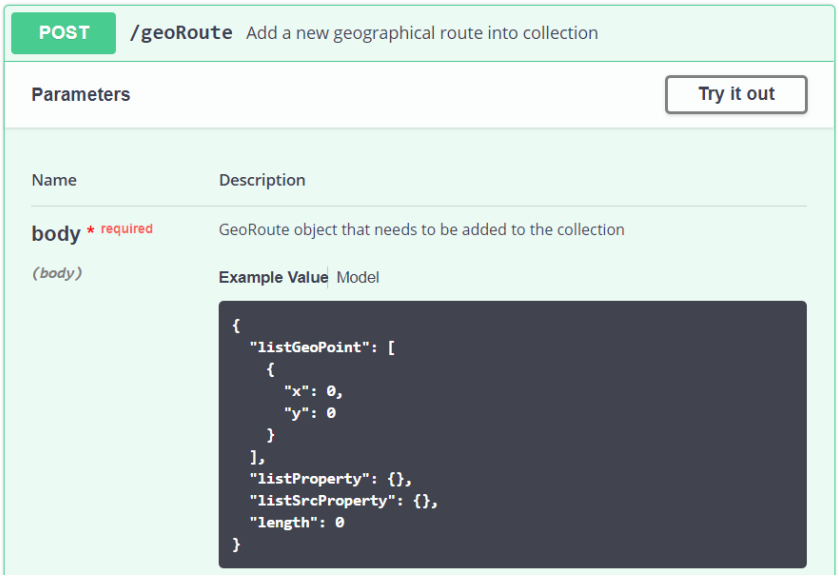


Fonte: autoria própria.

Como resposta da requisição, que foi realizada com sucesso, recebemos um objeto JSON contendo três atributos especiais. Todos os documentos contém estes atributos especiais, o identificador do documento, sua chave-primária e a revisão do documento. Na resposta

da requisição eles são identificados como: *_id, _key, _rev*, respectivamente. Além disso, no momento da criação de um *GeoRoute*, seus respectivos *InterGeoRoutes* são criados automaticamente. A Figura 10 exibe os detalhes da rota para adicionar uma nova aresta na coleção orientada a grafos *GeoRoute*.

Figura 10 – Detalhes da rota para criar um *GeoRoute*.



Fonte: autoria própria.

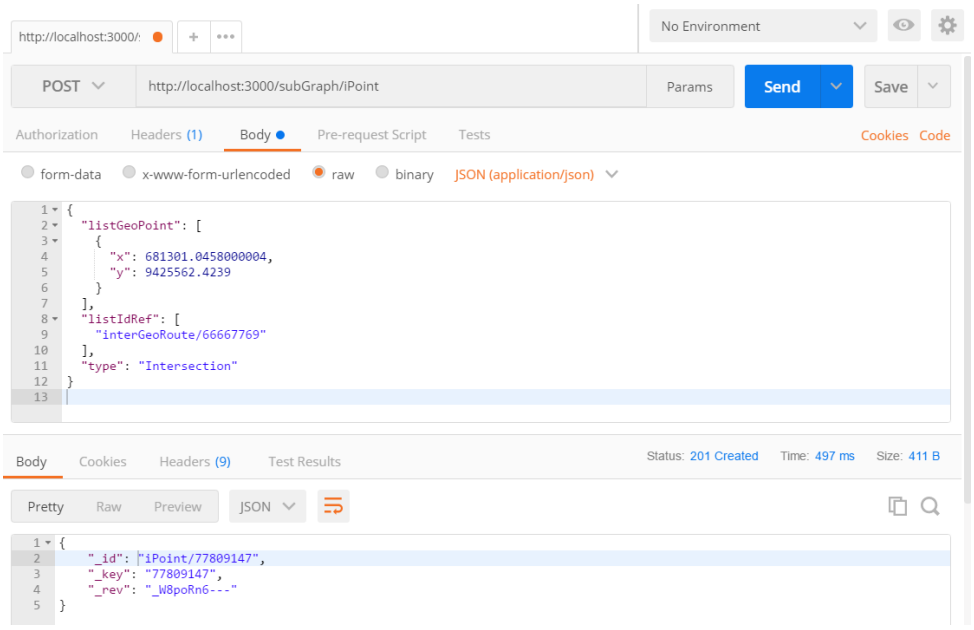
4.5 Criar IPoint

Os *IPoints* representam os pontos de interesse e, como já citado anteriormente, podem representar uma interseção, uma linha ou um polígono. Enviaremos, no corpo da mensagem, um objeto contendo os seguintes atributos: *listGeoPoint, listIdRef* e *type*. Para adicionar um novo *IPoint* utilizaremos o recurso *IPoint*.

Como já citado na seção anterior, o *listGeoPoint* contém as coordenadas geográficas. O *listIdRef* irá conter os valores do atributo *_id* de seus respectivos *InterGeoRoute* e o atributo *type* define o tipo do *IPoint* no qual estamos trabalhando (no nosso caso será *Intersection*).

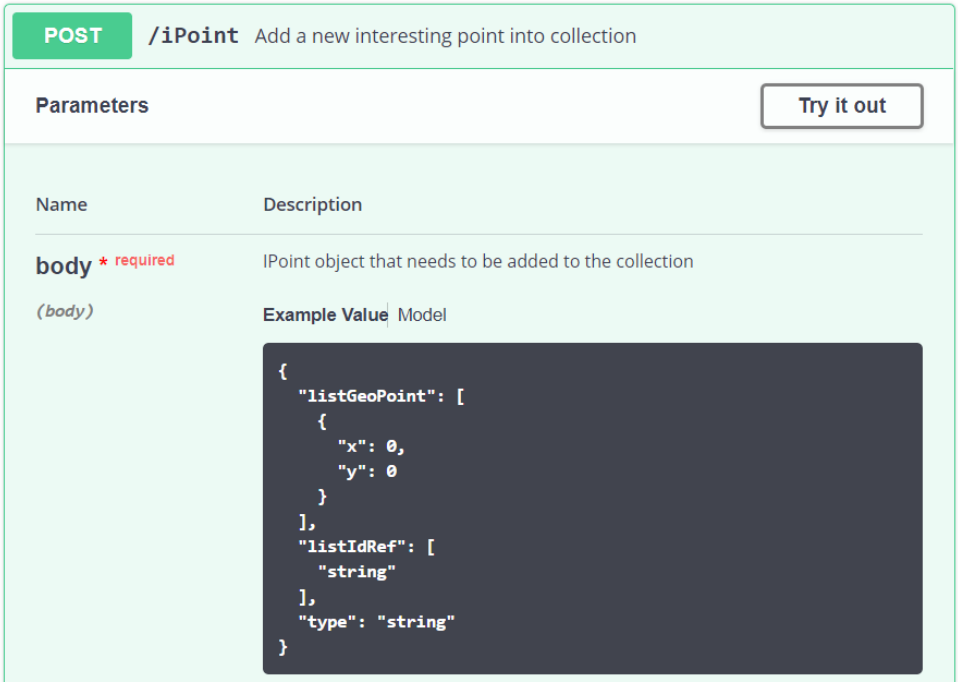
Como pode ser visto na Figura 11, ao concluir a requisição com sucesso, este recurso também retorna, somente, os atributos especiais que, automaticamente, foram criados pelo próprio ArangoDB. A Figura 12 exibe os detalhes da rota para a criação de um novo registro na coleção de *IPoint*.

Figura 11 – Resultado da adição de um novo *IPoint*.



Fonte: autoria própria.

Figura 12 – Detalhes da rota para criar um *IPoint*.



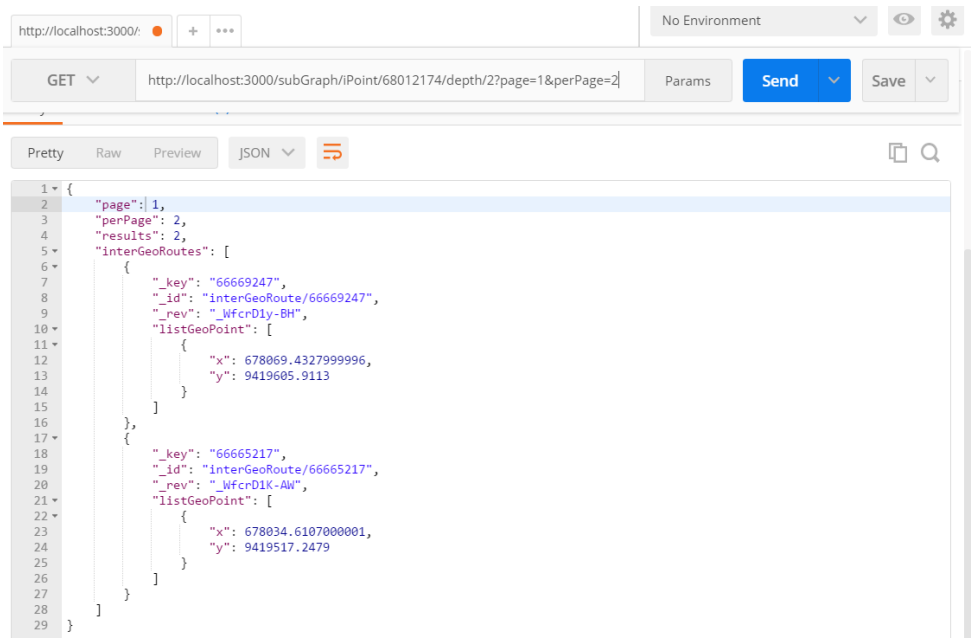
Fonte: autoria própria.

4.6 Busca Em Profundidade

Uma parte importante deste trabalho é lidar com estrutura de grafos, permitindo que os mais diversos algoritmos do ramo da teoria dos grafos possam ser executados em cima dos dados fornecidos pela API.

Uma das facilidades oferecidas pelo ArangoDB é a busca em profundidade. Para realizar uma busca é preciso informar um determinado *IPoint* e a profundidade da busca a ser executada. Utilizaremos o recurso *iPoint/:key/depth/:depth* passando os parâmetros *page* e *perPage* de modo que, tais parâmetros, irão determinar a página a ser exibida e a quantidade de resultados por página.

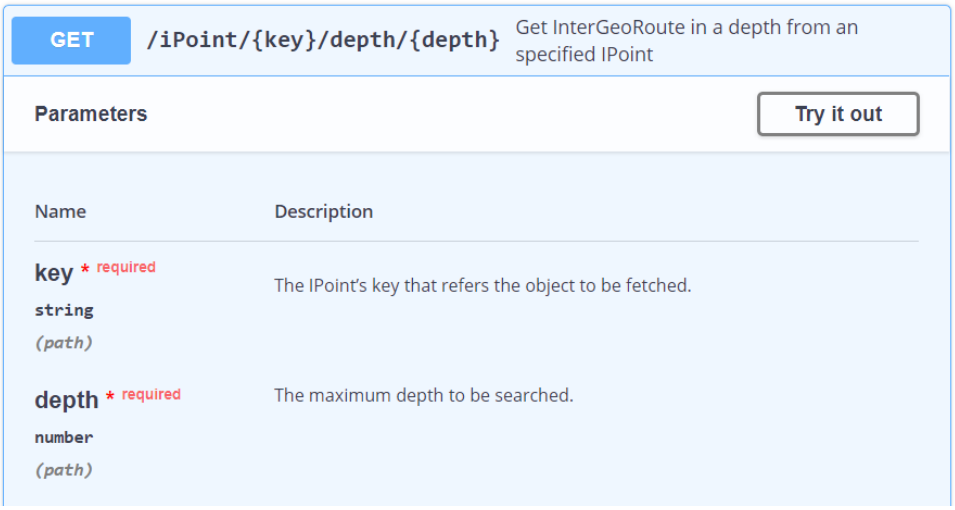
Figura 13 – Resultado da busca em profundidade.



Fonte: autoria própria.

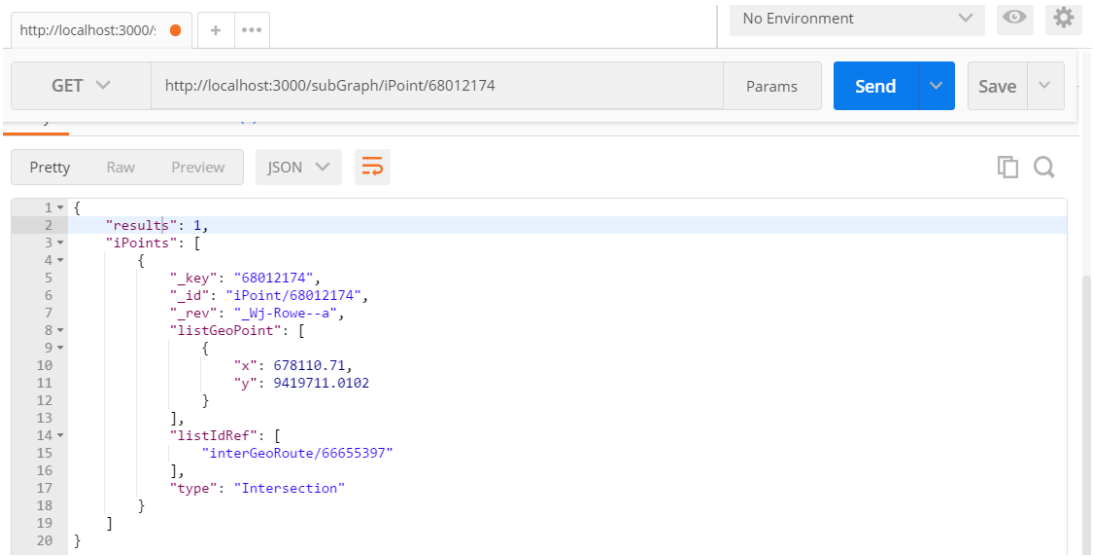
A Figura 13 exibe os dois primeiros resultados da busca em profundidade. Na rota, passamos a *_key* do *IPoint* 68012174 e, na profundidade, informamos o valor 2. Para entendermos melhor o resultado desta requisição, a Figura 15 exibe os detalhes do *IPoint* 68012174.

Figura 14 – Detalhes da rota para realizar uma busca em profundidade.



Fonte: autoria própria.

Figura 15 – Resultado da busca de um *IPoint* específico.



Fonte: autoria própria.

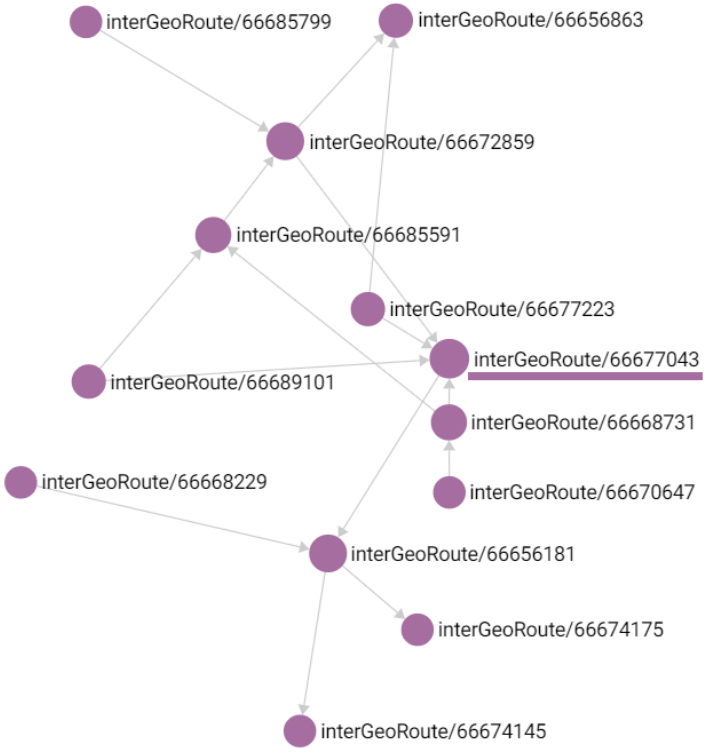
4.7 Resultados

Atualmente, a API, juntamente com o ArangoDB, manuseiam exatos 31.932 registros, sendo estes, arestas que representam todos os trechos de vias existentes na cidade de Mossoró, Rio Grande do Norte. O grafo apresentado na Figura 16, exibe uma representação a partir dos dados recebidos em resposta à requisição a rota de busca em profundidade, como o exemplo apresentado na Figura 13. Em destaque no grafo está o *InterGeoRoute* definido como ponto de partida. A Figura 17 exibe uma representação real do grafo da Figura 16, exibindo o trecho de via usado como base para a busca em profundidade e os resultados dos trechos com suas respectivas profundidades.

Com seu desenvolvimento concluído, a API apresentou um funcionamento pleno e ágil. Quando em ação, pôde-se verificar que todas as operações implementadas funcionam conforme o esperado. Além disso, a API permite às aplicações consumirem o *Web Service* de modo inteligente e eficiente, dispensando a necessidade de trabalhar em cima das informações relevantes, ao invés disso, cada ponto de interseção é convertido em um *InterGeoRoute* e, consequentemente, em um *IPoint*, permitindo que algoritmos possam ser executados com base em estruturas de dados mais compactas e, somente quando obtido um resultado, tenham suas referências retomadas.

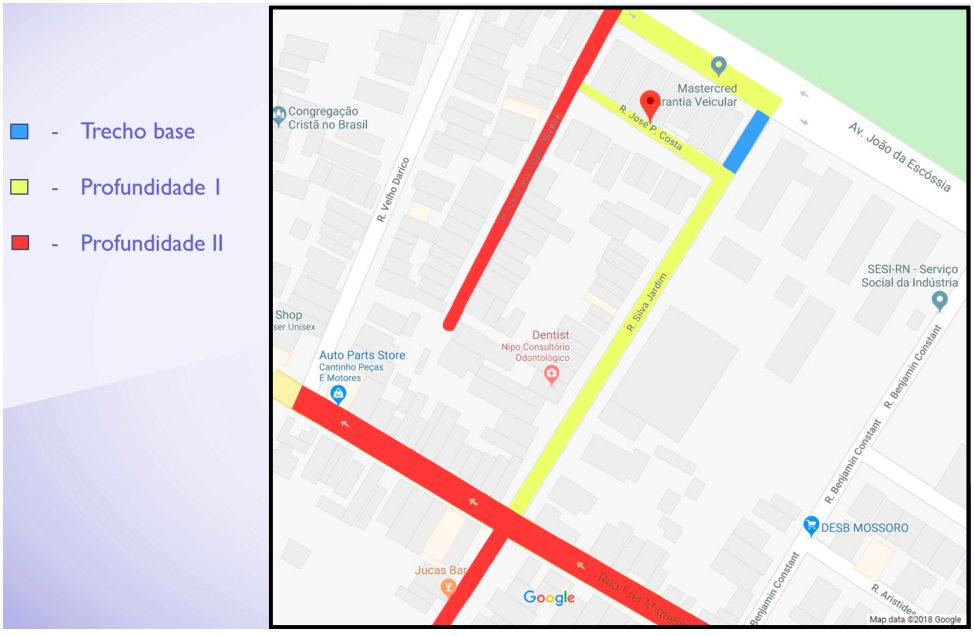
Ao todo são duas camadas que compõem a API e comunicam-se uma com a outra. A primeira camada, representada pelo banco de dados, é responsável pelo armazenamento de informações através de coleções e estruturas de grafos. A segunda camada, e o principal objeto de estudo deste trabalho, é a API em si, através da implementação do estilo arquitetural REST, ela é encarregada de intermediar as operações e consultas a serem executadas no banco de dados, não permitindo que procedimentos indesejáveis sejam executados.

Figura 16 – Grafo representando o resultado da busca em profundidade.



Fonte: autoria própria.

Figura 17 – Representação em mapa do grafo apresentado na Figura 16.



Fonte: autoria própria.

5 CONSIDERAÇÕES FINAIS

O principal objetivo deste trabalho era a criação e implementação de uma API baseada no estilo arquitetural REST. O intuito do uso de uma API era permitir que, diferentes aplicações, em diferentes linguagens, possam usufruir do uso informações relevantes sem a necessidade de criar diferentes *middlewares* para manusear os mais diversos tipos de dados. Através de uma API flexível, pode-se definir diferentes estruturas, capazes de atender os mais diversos tipos de problemas, simplificando e diminuindo custos nos processos de desenvolvimento de aplicações.

A implementação de uma API que segue os padrões do estilo arquitetural REST prova que está é a ferramenta ideal para a construção de serviços descentralizados, interoperáveis e distribuídos. Sua capacidade de unificar a comunicação com as diversas camadas de uma aplicação faz com seus recursos sejam facilmente implementados.

Como trabalhos futuros, poderíamos criar uma camada de *front-end*, simplificando para o usuário o consumo da API RESTful e a representação dos resultados. Com isso, alcançaríamos o feito de representar graficamente os dados, ao invés de apenas apresentar informações textuais em forma de JSON, como pôde ser visto anteriormente neste trabalho. Além disso, seria de grande valia concretizar um *middleware* responsável pela autenticação e controle de permissões de acesso, impedindo que usuários não autorizados executem operações críticas.

REFERÊNCIAS BIBLIOGRÁFICAS

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

ARANGODB. *ArangoDB*. 2017. Disponível em: <https://www.arangodb.com/>.

BOOTH, D. et al. *Web Services Architecture*. 2004. Disponível em: <https://www.w3.org/TR/ws-arch/>.

BUTLER, H. et al. *The GeoJSON Format*. 2016.

CENSIPAM. *Quantum GIS: guia do usuário*. [S.l.], 2010.

CLEMENT, S. *Introduction to Oracle NoSQL Database*. [S.l.]: Oracle, 2012.

COOKSEY, B. *An Introduction do APIs*. [S.l.]: Zapier, 2014.

COSTA, B. et al. Evaluating a representational state transfer (rest) architecture: What is the impact of rest in my architecture? In: *Proceedings of the 2014 IEEE/IFIP Conference on Software Architecture*. Washington, DC, USA: IEEE Computer Society, 2014. (WICSA '14), p. 105–114. ISBN 978-1-4799-3412-6. Disponível em: <https://doi.org/10.1109/WICSA.2014.29>.

DAL, T.; DORNELES, C.; REBONATTO, M. Web services ws-* versus web services rest. 01 2009.

EIFREM, E.; ROBINSON, I.; WEBBER, J. *Graph Databases: new opportunities for connected data*. 1005 Gravenstein Highway North, Sebastopol, CA 95472: O'Reilly Media, Inc., 2015.

FIELDING, R. T. *Architectural Styles and the Design of Network-based Software Architectures*. Tese (Doutorado) — University of California, Irvine, 2000.

GEOJSON. *GeoJSON*. 2017. Disponível em: <http://geojson.org>.

GERSTING, J. L. *Fundamentos Matemáticos Para a Ciência da Computação*. Rio de Janeiro: LTC - Livros Técnicos e Científicos Editora S.A, 2013.

HAMADA, R. R. do V. G. E. *Introdução ao Geoprocessamento: princípios básicos e aplicação*. [S.l.], 2007.

MENZORI, M. *Georreferenciamento: conceitos*. first. São Paulo: Editora Baraúna, 2017.

NOSQL. *NOSQL Databases*. 2009. Disponível em: <http://nosql-database.org>.

OLIVEIRA, P. H. C. de. *Desenvolvimento de Um Gerador de API REST Seguindo Os Principais Padrões da Arquitetura*. Marília, SP, 2014. Monografia, Centro Universitário Eurípodes de Marília.

OLIVEIRA Álvaro Gabriel Gomes de. *Construção de Aplicações Distribuídas Utilizando-se de APIs REST*. Mossoró, RN, 2018. Monografia, Universidade do Estado do Rio Grande do Norte.

PANIZ, D. *NoSQL: Como Armazenar os Dados de Uma Aplicação Moderna*. Rua Vergueiro, 3185 - 8º andar, 04101-300, Vila Mariana, São Paulo, SP, Brasil: Casa do Código, 2016.

SASAKI, B. M. *Graph Databases for Beginners: ACID vs. BASE explained*. 2015. Disponível em: <https://neo4j.com/blog/acid-vs-base-consistency-models-explained/>.

SILVA, C. B. da. *Um Modelo Computacional Para Integração de Problemas de Otimização Utilizando Banco de Dados Orientados a Grafos*. Mossoró, RN, 2017. Monografia, Universidade do Estado do Rio Grande do Norte.

SOUSA, F. P. de. *Criação de Framework REST/HATEOAS Open Source para desenvolvimento de APIs em NodeJS*. Dissertação (Mestrado) — Faculdade de Engenharia da Universidade do Porto, 2015.

STEIN, C.; DRYSDALE, R. L.; BOGART, K. *Matemática Discreta Para Ciência da Computação*. São Paulo: Pearson Education do Brasil, 2013.

SUCUPIRA, I. R. *Um Estudo Empírico de Hiper-Heurísticas*. Dissertação (Mestrado) — Universidade de São Paulo, 2007.

Anexos

ANEXO A – RECURSOS E ROTAS

1. Definitions

1.1. PaginatedAbsGeoRoute

Name	Type	Description	Required
page	integer		No
perPage	integer		No
results	integer		No
absGeoRoutes	array	See **AbsGeoRoute** in the **Definitions** section.	No

1.2. PaginatedGeoRoute

Name	Type	Description	Required
page	integer		No
perPage	integer		No
results	integer		No
geoRoutes	array	See **GeoRoute** in the **Definitions** section.	No

1.3. PaginatedInterGeoRoute

Name	Type	Description	Required
page	integer		No
perPage	integer		No
results	integer		No
interGeoRoutes	array	See **InterGeoRoute** in the **Definitions** section.	No

1.4. PaginatedIPoint

Name	Type	Description	Required
page	integer		No
perPage	integer		No
results	integer		No
iPoints	array	See **IPoint** in the **Definitions** section.	No

1.5. AbsGeoRoute

Name	Type	Description	Required
_from	string		No
_to	string		No
length	number		No

1.6. GeoPoint

Name	Type	Description	Required
x	number		No

Name	Type	Description	Required
y	number		No

1.7. GeoRoute

Name	Type	Description	Required
listGeoPoint	array	See **GeoPoint** in the **Definitions** section.	No
listProperty	object		No
listSrcProperty	object		No
length	number		No

1.8. InterGeoRoute

Name	Type	Description	Required
listGeoPoint	array	See **GeoPoint** in the **Definitions** section.	No

1.9. IPoint

Name	Type	Description	Required
listGeoPoint	array	See **GeoPoint** in the **Definitions** section.	No
listIdRef	array		No
type	string		No

2. Paths

2.2 get /absGeoRoute

Summary	Get all existings AbsGeoRoute
Description	
Operation Id	getAllAbs
Produces	application/json
Consumes	

Parameters

Name	In	Description	Required	Type	Format	Collection Format	Default	Min	Max
page	query	The page number to start at. Default is 1.	No	integer	int32				
perPage	query	How many items should be displayed in a single page. Default is 10.	No	integer	int32				

Tags AbsGeoRoute

Responses

code	description
200	successful operation

See **PaginatedAbsGeoRoute** in the **Definitions** section.

Name	Type	Required
page	integer	No
perPage	integer	No
results	integer	No
absGeoRoutes	array	No

2.2 post /absGeoRoute

Summary	Add a new abstract route into collection
Description	
Operation Id	createAbs
Produces	application/json
Consumes	application/json

Parameters

Name	In	Description	Required	Type	Format	Collection Format	Default	Min	Max

Name	In	Description	Required	Type	Format	Collection Format	Default	Min	Max
body	body	AbsGeoRoute object that needs to be added to the collection Name / Type / Required _from / string / No _to / string / No length / number / No	Yes						

Tags AbsGeoRoute

Responses

code	description
405	Invalid input

2.3 get /absGeoRoute/{key}

Summary Filter AbsGeoRoute by key

Description

Operation Id filterByKeyAbs

Produces application/json

Consumes

Parameters

Name	In	Description	Required	Type	Format	Collection Format	Default	Min	Max
key	path	The AbsGeoRoute's key that needs to be fetched.	Yes	string					

Tags AbsGeoRoute

Responses

code	description
200	successful operation See **AbsGeoRoute** in the **Definitions** section. Name / Type / Required _from / string / No _to / string / No length / number / No
400	Invalid AbsGeoRoute supplied
404	AbsGeoRoute not found

2.3 put /absGeoRoute/{key}

Summary Update an existing AbsGeoRoute.

Description

Operation Id updateAbs

Produces application/json

Consumes

Parameters

Name	In	Description	Required	Type	Format	Collection Format	Default	Min	Max

Name	In	Description	Required	Type	Format	Collection Format	Default	Min	Max
key	path	The AbsGeoRoute's key that refers the object to be updated.	Yes	string					
body	body	Updated AbsGeoRoute object Name / Type / Required _from / string / No _to / string / No length / number / No	Yes						

Tags AbsGeoRoute

Responses

code	description
400	Invalid AbsGeoRoute supplied
404	AbsGeoRoute not found

2.3 delete /absGeoRoute/{key}

Summary Delete an existing AbsGeoRoute.

Description

Operation Id deleteAbs

Produces application/json

Consumes

Parameters

Name	In	Description	Required	Type	Format	Collection Format	Default	Min	Max
key	path	The AbsGeoRoute's key that refers the object to be deleted.	Yes	string					

Tags AbsGeoRoute

Responses

code	description
400	Invalid AbsGeoRoute supplied
404	AbsGeoRoute not found

2.4 get /absGeoRoute/{key}/geoRoute

Summary Return all the GeoRoute from an specified AbsGeoRoute

Description

Operation Id getGeoRouteByAbsGeoRouteKey

Produces application/json

Consumes

Parameters

Name	In	Description	Required	Type	Format	Collection Format	Default	Min	Max
key	path	The AbsGeoRoute's key from where GeoRoutes needs to be fetched.	Yes	string					

Name	In	Description	Required	Type	Format	Collection Format	Default	Min	Max
page	query	The page number to start at. Default is 1.	No	integer	int32				
perPage	query	How many items should be displayed in a single page. Default is 10.	No	integer	int32				

Tags AbsGeoRoute

Responses

code	description
200	successful operation See **PaginatedGeoRoute** in the **Definitions** section.

Name	Type	Required
page	integer	No
perPage	integer	No
results	integer	No
geoRoutes	array	No

400	Invalid AbsGeoRoute supplied
404	AbsGeoRoute not found

2.5 get /absGeoRoute/{key}/interGeoRoute

Summary Get all the InterGeoRoute from an specified AbsGeoRoute

Description

Operation Id getInterGeoRouteByAbsGeoRouteKey

Produces application/json

Consumes

Parameters

Name	In	Description	Required	Type	Format	Collection Format	Default	Min	Max
key	path	The AbsGeoRoute's key from where InterGeoRoute needs to be fetched.	Yes	string					
page	query	The page number to start at. Default is 1.	No	integer	int32				
perPage	query	How many items should be displayed in a single page. Default is 10.	No	integer	int32				

Tags AbsGeoRoute

Responses

code	description
200	successful operation

See **PaginatedIPoint** in the **Definitions** section.

Name	Type	Required
page	integer	No
perPage	integer	No
results	integer	No
iPoints	array	No

400	Invalid AbsGeoRoute supplied
404	AbsGeoRoute not found

2.6 get /absGeoRoute/{key}/iPoint

Summary	Return all the IPoint from an specified AbsGeoRoute
Description	
Operation Id	getIPointByAbsGeoRouteKey
Produces	application/json
Consumes	

Parameters

Name	In	Description	Required	Type	Format	Collection Format	Default	Min	Max
key	path	The AbsGeoRoute's key from where IPoints needs to be fetched.	Yes	string					
page	query	The page number to start at. Default is 1.	No	integer	int32				
perPage	query	How many items should be displayed in a single page. Default is 10.	No	integer	int32				

Tags AbsGeoRoute

Responses

code	description
200	successful operation

See **PaginatedIPoint** in the **Definitions** section.

Name	Type	Required
page	integer	No
perPage	integer	No
results	integer	No
iPoints	array	No

400	Invalid AbsGeoRoute supplied
404	AbsGeoRoute not found

2.7 get /geoRoute

Summary	Get all existings GeoRoute
Description	

Operation Id	getAllGeo
Produces	application/json
Consumes	

Parameters

Name	In	Description	Required	Type	Format	Collection Format	Default	Min	Max
page	query	The page number to start at. Default is 1.	No	integer	int32				
perPage	query	How many items should be displayed in a single page. Default is 10.	No	integer	int32				

Tags GeoRoute

Responses

code	description
200	successful operation

See **PaginatedGeoRoute** in the **Definitions** section.

Name	Type	Required
page	integer	No
perPage	integer	No
results	integer	No
geoRoutes	array	No

2.7 post /geoRoute

Summary	Add a new geographical route into collection
Description	
Operation Id	createGeo
Produces	application/json
Consumes	application/json

Parameters

Name	In	Description	Required	Type	Format	Collection Format	Default	Min	Max
body	body	GeoRoute object that needs to be added to the collection	Yes						

Name	Type	Required
listGeoPoint	array	No
listProperty	object	No
listSrcProperty	object	No
length	number	No

Tags GeoRoute

Responses

code	description
405	Invalid input

2.8 get /geoRoute/{key}

Summary	Filter GeoRoute by key
Description	
Operation Id	filterByKeyGeo
Produces	application/json
Consumes	

Parameters

Name	In	Description	Required	Type	Format	Collection Format	Default	Min	Max
key	path	The GeoRoute's key that needs to be fetched.	Yes	string					

Tags GeoRoute

Responses

code	description
200	successful operation

See **GeoRoute** in the **Definitions** section.

Name	Type	Required
listGeoPoint	array	No
listProperty	object	No
listSrcProperty	object	No
length	number	No

400	Invalid GeoRoute supplied
404	GeoRoute not found

2.8 put /geoRoute/{key}

Summary	Update an existing GeoRoute.
Description	
Operation Id	updateGeo
Produces	application/json
Consumes	

Parameters

Name	In	Description	Required	Type	Format	Collection Format	Default	Min	Max
key	path	The GeoRoute's key that refers the object to be updated.	Yes	string					
body	body	Updated GeoRoute object	Yes						

Name	Type	Required
listGeoPoint	array	No
listProperty	object	No
listSrcProperty	object	No
length	number	No

Name	In	Description	Required	Type	Format	Collection Format	Default	Min	Max

Name	Type	Required

Tags GeoRoute

Responses

code	description
400	Invalid GeoRoute supplied
404	GeoRoute not found

2.8 delete /geoRoute/{key}

Summary	Delete an existing GeoRoute.
Description	
Operation Id	deleteGeo
Produces	application/json
Consumes	

Parameters

Name	In	Description	Required	Type	Format	Collection Format	Default	Min	Max
key	path	The GeoRoute's key that refers the object to be deleted.	Yes	string					

Tags GeoRoute

Responses

code	description
400	Invalid GeoRoute supplied
404	GeoRoute not found

2.9 get /interGeoRoute

Summary	Get all existings InterGeoRoute
Description	
Operation Id	getAllInter
Produces	application/json
Consumes	

Parameters

Name	In	Description	Required	Type	Format	Collection Format	Default	Min	Max
page	query	The page number to start at. Default is 1.	No	integer	int32				
perPage	query	How many items should be displayed in a single page. Default is 10.	No	integer	int32				

Tags InterGeoRoute

Responses

code	description
200	successful operation

See **PaginatedInterGeoRoute** in the **Definitions** section.

Name	Type	Required
page	integer	No
perPage	integer	No
results	integer	No
interGeoRoutes	array	No

2.9 post /interGeoRoute

Summary	Add a new geographical intersection route into collection
Description	
Operation Id	createInter
Produces	application/json
Consumes	application/json

Parameters

Name	In	Description	Required	Type	Format	Collection Format	Default	Min	Max
body	body	InterGeoRoute object that needs to be added to the collection **Name** / **Type** / **Required** listGeoPoint / array / No	Yes						

Tags InterGeoRoute

Responses

code	description
405	Invalid input

2.10 get /interGeoRoute/{key}

Summary	Filter InterGeoRoute by key
Description	
Operation Id	filterByKeyInter
Produces	application/json
Consumes	

Parameters

Name	In	Description	Required	Type	Format	Collection Format	Default	Min	Max
key	path	The InterGeoRoute's key that needs to be fetched.	Yes	string					

Tags InterGeoRoute

Responses

code	description
200	successful operation See **InterGeoRoute** in the **Definitions** section. **Name** / **Type** / **Required** listGeoPoint / array / No
400	Invalid InterGeoRoute supplied
404	InterGeoRoute not found

2.10 put /interGeoRoute/{key}

Summary	Update an existing InterGeoRoute.
Description	
Operation Id	updateInter

Produces	application/json
Consumes	

Parameters

Name	In	Description	Required	Type	Format	Collection Format	Default	Min	Max
key	path	The InterGeoRoute's key that refers the object to be updated.	Yes	string					
body	body	Updated InterGeoRoute object **Name** / **Type** / **Required** listGeoPoint / array / No	Yes						

Tags	InterGeoRoute

Responses

code	description
400	Invalid InterGeoRoute supplied
404	InterGeoRoute not found

2.10 delete /interGeoRoute/{key}

Summary	Delete an existing InterGeoRoute.
Description	
Operation Id	deleteInter
Produces	application/json
Consumes	

Parameters

Name	In	Description	Required	Type	Format	Collection Format	Default	Min	Max
key	path	The InterGeoRoute's key that refers the object to be deleted.	Yes	string					

Tags	InterGeoRoute

Responses

code	description
400	Invalid InterGeoRoute supplied
404	InterGeoRoute not found

2.11 get /iPoint

Summary	Get all existings IPoint
Description	
Operation Id	getAllIPoint
Produces	application/json
Consumes	

Parameters

Name	In	Description	Required	Type	Format	Collection Format	Default	Min	Max
page	query	The page number to start at. Default is 1.	No	integer	int32				

Name	In	Description	Required	Type	Format	Collection Format	Default	Min	Max
perPage	query	How many items should be displayed in a single page. Default is 10.	No	integer	int32				

Tags IPoint

Responses

code	description
200	successful operation

See **PaginatedIPoint** in the **Definitions** section.

Name	Type	Required
page	integer	No
perPage	integer	No
results	integer	No
iPoints	array	No

2.11 post /iPoint

Summary Add a new interesting point into collection

Description

Operation Id createIPoint

Produces application/json

Consumes application/json

Parameters

Name	In	Description	Required	Type	Format	Collection Format	Default	Min	Max
body	body	IPoint object that needs to be added to the collection Name / Type / Required listGeoPoint / array / No listIdRef / array / No type / string / No	Yes						

Tags IPoint

Responses

code	description
405	Invalid input

2.12 get /iPoint/{key}

Summary Filter IPoint by key

Description

Operation Id filterByKeyIPoint

Produces application/json

Consumes

Parameters

Name	In	Description	Required	Type	Format	Collection Format	Default	Min	Max
key	path	The IPoint's key that needs to be fetched.	Yes	string					

Tags IPoint

Responses

code	description
200	successful operation See **IPoint** in the **Definitions** section. **Name** / **Type** / **Required** listGeoPoint / array / No listIdRef / array / No type / string / No
400	Invalid IPoint supplied
404	IPoint not found

2.12 put /iPoint/{key}

Summary Update an existing IPoint.

Description

Operation Id updateIPoint

Produces application/json

Consumes

Parameters

Name	In	Description	Required	Type	Format	Collection Format	Default	Min	Max
key	path	The IPoint's key that refers the object to be updated.	Yes	string					
body	body	Updated IPoint object **Name** / **Type** / **Required** listGeoPoint / array / No listIdRef / array / No type / string / No	Yes						

Tags IPoint

Responses

code	description
400	Invalid IPoint supplied
404	IPoint not found

2.12 delete /iPoint/{key}

Summary Delete an existing IPoint.

Description

Operation Id deleteIPoint

Produces application/json

Consumes

Parameters

Name	In	Description	Required	Type	Format	Collection Format	Default	Min	Max
key	path	The IPoint's key that refers the object to be deleted.	Yes	string					

Tags IPoint

Responses

code	description
400	Invalid IPoint supplied
404	IPoint not found

2.13 get /iPoint/{key}/interGeoRoute

Summary Get InterGeoRoute from an specified IPoint

Description

Operation Id getInterGeoRouteByIPointKey

Produces

Consumes

Parameters

Name	In	Description	Required	Type	Format	Collection Format	Default	Min	Max
key	path	The IPoint's key that refers the object to be fetched.	Yes	string					

Tags IPoint

Responses

code	description
200	successful operation See **PaginatedInterGeoRoute** in the **Definitions** section.
400	Invalid IPoint supplied
404	IPoint not found

Name	Type	Required
page	integer	No
perPage	integer	No
results	integer	No
interGeoRoutes	array	No

2.14 get /iPoint/{key}/depth/{depth}

Summary Get InterGeoRoute in a depth from an specified IPoint

Description

Operation Id getIntersectionByDepth

Produces

Consumes

Parameters

Name	In	Description	Required	Type	Format	Collection Format	Default	Min	Max

Name	In	Description	Required	Type	Format	Collection Format	Default	Min	Max
key	path	The IPoint's key that refers the object to be fetched.	Yes	string					
depth	path	The maximum depth to be searched.	Yes	number					

Tags IPoint

Responses

code	description
200	successful operation

See **PaginatedInterGeoRoute** in the **Definitions** section.

Name	Type	Required
page	integer	No
perPage	integer	No
results	integer	No
interGeoRoutes	array	No

400	Invalid IPoint supplied
404	IPoint not found

2.15 get /iPoint/type/{type}

Summary Get a random IPoint from an specified IPoint's type

Description

Operation Id random

Produces

Consumes

Parameters

Name	In	Description	Required	Type	Format	Collection Format	Default	Min	Max
type	path	The IPoint's type to be fetched.	Yes	string					

Tags IPoint

Responses

code	description
200	successful operation

See **PaginatedIPoint** in the **Definitions** section.

Name	Type	Required
page	integer	No
perPage	integer	No
results	integer	No
iPoints	array	No

400	Invalid IPoint's type supplied
404	IPoint's type not found