UNIVERSIDADE FEDERAL DO ESTADO DO RIO DE JANEIRO CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIA ESCOLA DE INFORMÁTICA APLICADA

DESENVOLVIMENTO DE UMA API DE APOIO À MOBILIDADE URBANA

Bruno Sales Domingos da Silva

Orientador

Carlos Alberto Campos

RIO DE JANEIRO, RJ - BRASIL
FEVEREIRO DE 2021

Catalogação informatizada pelo autor

S586

Silva, Bruno Sales Domingos da
Desenvolvimento de uma api de apoio à mobilidade urbana / Bruno Sales Domingos da Silva. -- Rio de Janeiro, 2021.
67 f

Orientador: Carlos Alberto Campos.
Trabalho de Conclusão de Curso (Graduação) - Universidade Federal do Estado do Rio de Janeiro, Graduação em Sistemas de Informação, 2021.

1. API. 2. Mobilidade urbana. 3. Transporte público. 4. GPS. 5. ETL. I. Campos, Carlos Alberto, orient. II. Título.

DESENVOLVIMENTO DE UMA API DE APOIO À MOBILIDADE URBANA

Bruno Sales Domingos da Silva

Projeto de Graduação apresentado à Escola de Informática Aplicada da Universidade Federal do Estado do Rio de Janeiro (UNIRIO) para obtenção do título de Bacharel em Sistemas de Informação.

Aprovada por:

_____

Carlos Alberto Campos - UNIRIO

_____

Morganna Carmem Diniz - UNIRIO

_____

Elton Figueiredo de Souza Soares - UNIRIO

RIO DE JANEIRO, RJ - BRASIL
FEVEREIRO DE 2021

RESUMO

Com grande parte da população mundial vivendo em áreas urbana, diversos grandes desafios gerenciais surgiram principalmente em cidades de grande perfil demográfico, sendo um desses desafios a mobilidade urbana. O número crescente de veículos nas ruas vem causando preocupação com emissões de poluentes na atmosfera e congestionamentos de trânsito, impactando até mesmo na qualidade de vida das pessoas. Todos estes fatores fazem com que transportes públicos tenham sua importância elevada e, neste contexto, se faz necessário a presença de sistemas de informação que apoiem a sociedade em seu dia a dia durante seu deslocamento. Este trabalho apresenta o desenvolvimento de uma Interface de Programação de Aplicativos (API) que disponibiliza dados atualizados sobre geolocalização de transportes públicos e informações de itinerários, linhas e modais. Para mostrar a aplicabilidade da ferramenta é feito um estudo de caso da cidade do Rio de Janeiro.

Palavras-chave: API, mobilidade urbana, transporte público, GPS, ETL.

ABSTRACT

Most of the global population currently lives in urban areas, and consequently, there are several management challenges that need to be addressed, especially in cities with a large demographic profile. One of these challenges is urban mobility. The growing number of vehicles on the streets has been causing concern about pollutant emissions into the atmosphere and traffic congestion, impacting people's quality of life. All of these factors make public transport extremely important, as the need of information systems, which provides support to society in it's daily commute. This work presents the development of an Application Programming Interface (API) that provides updated data about public transport geolocation, as well routes, lines and modals. A case study was conducted in the city of Rio de Janeiro in order to show the API's applicability.

Keywords: API, urban mobility, public transport, GPS, ETL.

Sumário

1 Introdução **1**

1.1 Motivação 1

1.2 Objetivo 3

1.3 Organização do texto 4

2 Fundamentação Teórica **5**

2.1 ETL 5

2.2 Web API 6

2.2.1 Arquitetura REST 6

2.3 Aplicações relacionadas de apoio à mobilidade urbana 8

2.3.1 Google Maps 8

2.3.2 Moovit 9

2.3.3 Citymapper 9

2.3.4 Rio.bus 10

2.4 Considerações finais 10

3 A aplicação Ikuzo API **11**

3.1 Arquitetura 11

3.2 Modelagem de dados 11

3.2.1 Classes de domínio 12
3.2.2 Banco de dados 13
3.3 Implementação da aplicação 16
3.3.1 Arquitetura de desenvolvimento 16
3.3.2 Conexão ao banco de dados 17
3.3.3 Segurança e autenticação 17
3.3.4 Definição de recursos da aplicação 18
3.3.5 Modelagem de sequência 20
3.4 Considerações finais 31

4 Estudo de caso **32**

4.1 Repositórios de dados 32
4.1.1 Data.Rio 32
4.1.2 Rio.bus 33
4.2 Atualização de dados 33
4.3 Mapeamentos 36
4.3.1 Mapeamento de campos: Rio.bus 36
4.3.2 Mapeamento de campos: Data.rio 36
4.3.3 Campos calculados 37
4.4 A aplicação Ônibus Carioca 37
4.4.1 Requisições à Ikuzo API 38
4.4.2 Shared Preferences 38
4.4.3 Permissionamento 38
4.4.4 Arquitetura de desenvolvimento 38
4.4.5 Funcionalidades da aplicação Ônibus Carioca 39

4.5 Resultados obtidos e discussões 41
4.5.1 Consultas simples de dados 41
4.5.2 Consultas georreferenciadas 45
4.5.3 Resultados quantitativos 48
4.5.4 Estudos em dados históricos 49
4.6 Considerações finais 53

5 Conclusão **55**

5.1 Considerações gerais 55
5.2 Trabalhos futuros 56

Lista de Figuras

2.1 O processo de ETL 5

3.1 Arquitetura geral proposta 12

3.2 Modelagem de classes de domínio da Ikuzo API 13

3.3 Modelo relacional 15

3.4 Arquitetura de camadas na Ikuzo API e sua interdependência 17

3.5 Modelagem de sequência para atualização de linhas 21

3.6 Modelagem de sequência para atualização de itinerários 22

3.7 Modelagem de sequência para atualização de modais 23

3.8 Modelagem de sequência para atualização de GPS 24

3.9 Modelagem de sequência para obtenção de rota 25

3.10 Modelagem de sequência para listagem de linhas 26

3.11 Modelagem de sequência para detalhes de uma linha 26

3.12 Modelagem de sequência para listagem de itinerários 27

3.13 Modelagem de sequência para obtenção de linhas de um local 28

3.14 Modelagem de sequência para listagem de modais 29

3.15 Modelagem de sequência para detalhes de um modal 30

3.16 Modelagem de sequência para obtenção de modais próximos a uma latitude e longitude 31

4.1 Arquitetura entre as aplicações 33

4.2 Programa em C# para chamada por minuto do recurso da Ikuzo API de atualização de GPS 35

4.3 Classe C# para requisição REST do recurso da Ikuzo API de atualização de GPS 35

4.4 Arquitetura entre os sistemas integrados 37

4.5 Diagrama de classes da aplicação Ônibus Carioca 39

4.6 Menu de navegação da aplicação Ônibus Carioca 40

4.7 Configurações da aplicação Ônibus Carioca 41

4.8 Busca de linhas na Ikuzo API 42

4.9 Busca de uma linha específica na Ikuzo API 43

4.10 Busca de itinerário de uma linha na Ikuzo API 43

4.11 Busca de ônibus na Ikuzo API 44

4.12 Busca de um ônibus específico na Ikuzo API 45

4.13 Tela com ônibus próximos ao ponto 46

4.14 Tela de linhas que possuem itinerário próximo ao ponto 47

4.15 Busca de planejamento de rota na Ikuzo API 48

4.16 Pontos de GPS coletado da linha 107 no período analisado 50

4.17 Itinerário completo cadastrado da linha 107 50

4.18 Itinerário completo cadastrado da linha 774 52

4.19 Itinerário completo cadastrado da linha 775 52

Lista de Tabelas

3.1 Definição de recursos da Ikuzo API 19

4.1 Mapeamentos realizados sobre dados do repositório Rio.bus 36

4.2 Mapeamentos realizados sobre dados do repositório Data.rio 36

4.3 Tempo médio para atualização dos recursos 48

4.4 Horários e tempos de viagem do modal A63536 da linha 107 51

4.5 Horários e tempos de viagem do modal C27120 da linha 774 53

4.6 Horários e tempos de viagem do modal C27046 da linha 775 53

1. Introdução

Neste capítulo são discutidas as motivações para o desenvolvimento do trabalho, os objetivos a serem alcançados, além da organização do texto nessa monografia.

1.1 Motivação

Estima-se que 54% da população mundial vive em áreas urbana. Acredita-se que, até 2050, este número deverá chegar a 66%, de acordo com o relatório *World Urbanizacion Prospects* da ONU [1]. Se levarmos em consideração apenas o Brasil, 85% da população brasileira vive em centros urbanos, segundo o censo do IBGE [2]. Estes números trazem consigo grandes desafios gerenciais em cidades, e um deles é a mobilidade urbana.

Com cada vez mais pessoas precisando se locomover em grandes cidades é natural pensarmos em trânsito, e é neste âmbito que os meios de transporte públicos se tornam imprescindíveis ao visarmos a redução do número de veículos automotores, consequentemente, a desobstrução de vias, redução de congestionamentos e, finalmente, redução de emissão de poluentes no ambiente, elevando por assim, a qualidade de vida da população que precisa se deslocar diariamente para suas atividades [3].

A otimização da gestão de trânsito e transporte em grandes metrópoles pode ser viabilizada por meio da utilização de sistemas online integrados, compostos de diferentes tecnologias, direcionados a atuar para controle e melhoria da mobilidade, sendo chamados de Sistemas de Transporte Inteligentes (*Intelligent Transportation Systems* – ITS).

Os ITS devem oferecer uma ligação inteligente entre os usuários dos sistemas de transportes, os veículos e a infraestrutura. Para isso utilizam uma ampla variedade de tecnologias tais como: processamento de informação e comunicação, sensoriamento, navegação e tecnologia de controle aplicados a melhoria do gerenciamento e operação dos sistemas de transportes, a fim de otimizar a eficiência no uso das vias, a segurança viária, através

de redução de tempos de espera e dos impactos ambientais.

Uma maneira de ver ITS é como uma aplicação de uso intensivo de dados, em que os dados são hospedados e circulam por meio de uma rede interconectada de computadores, infraestrutura de comunicação e infraestrutura de transporte. Este sistema é caracterizado por: (1) produtores e consumidores de dados, (2) sistemas de armazenamento de dados e (3) componentes de suporte inteligente à decisão. A comunicação é suportada por tecnologias com e sem fio. Através da rede, aplicativos de suporte à decisão extraem dados relevantes que são produzidos por milhares de fontes, especificamente de sensores rodoviários e dispositivos ITS. Os dados são então usados para fornecer serviços específicos para usuários de estradas, planejadores de transporte e formuladores de políticas.

Uma segunda maneira de entender o ITS envolve considerar as várias camadas da arquitetura. Para este sistema, a camada de base contém os componentes físicos de transporte, redes de computadores, computadores e dispositivos de armazenamento. Esses componentes de computação podem ser *commodities* prontos para uso ou podem ser especificamente dispositivos proprietários que são usados por uma pequena comunidade ou uma única empresa. O sistema é também caracterizado por uma série de padrões definidos que permitem que as redes se conectem a computadores e a dispositivos de armazenamento.

Acima da camada física fundamental está a camada de enlace de dados, que é caracterizada por uma série de padrões cada vez mais sofisticados que definem protocolos de comunicação para tecnologias de rede específicas, como redes sem fio ou com fio. O protocolo de internet (IP), que é o protocolo padrão para conectar diferentes redes, funciona acima dos protocolos individuais de rede para permitir a comunicação veicular via telefone celular para um data center que é interconectado com tecnologias de rede com fio, como Ethernet 10G. Protocolos da camada de transporte acima do IP, como o protocolo TCP e outros garantem uma confiabilidade de ponta a ponta de comunicação mesmo quando as diferentes fontes estão se movendo e mudando. Os protocolos da camada de sessão, apresentação e aplicação acima da camada de transporte descrevem os formatos de dados esperados pelos aplicativos, em seguida, gerencia os diferentes tipos de mensagens comunicadas entre os usuários e sistemas e entre diferentes sistemas autônomos [5].

Os aplicativos de mobilidade ITS têm a finalidade de fornecer serviços de mobilidade, como a rota mais curta entre uma origem e o destino considerando diferentes fatores (por exemplo, distância, tempo, consumo de energia) em um ambiente de viagem rico em dados baseado em informações coletadas pelas tecnologias de coleta de dados ITS.

Ajustando os sinais de tráfego, gerenciando dinamicamente as operações de trânsito ou despachando serviços de manutenção de emergência, esses aplicativos podem ajudar os centros de gerenciamento de transporte a monitorar e gerenciar o desempenho do sistema de transporte [5].

Um grande desafio em ITS está em como processar e modelar grandes volumes de dados não estruturados para análise posterior, que não pode ser tratada de forma eficaz por abordagens tradicionais. Há necessidade de desenvolvimento de serviços e APIs capazes de processar e inferir informações em tempo real para melhor apoiar a tomada de decisão. Outros aspectos que devem ser considerados são os desafios impulsionados por um contexto de inconsistências nos próprios dados, dados desatualizados, a largura de banda da conexão. Essas inconsistências levam à falta de qualidade nos conjuntos de dados, acrescentando a isso, o desafio de fundir e harmonizar grandes volumes de dados de muitas fontes ao mesmo tempo.

A fim de enfrentar os desafios declarados anteriormente, os dados precisam ser coletados, limpos, transformados e armazenados eficientemente. Só então é possível extrair o valor de uma grande quantidade de dados. O conceito extração, transformação e carga (*Extract, Transform, Load* - ETL) é uma abordagem possível para lidar com os aspectos supracitados.

1.2 Objetivo

Este trabalho tem por objetivo o desenvolvimento de uma Interface de Programação de Aplicativos (*Application Programming Interface* – API) de apoio à mobilidade urbana. Se integrando a repositórios públicos e realizando processos de extração, transformação e carga (*Extract, Transform, Load* - ETL) em banco de dados.

Adicionalmente, também faremos o desenvolvimento de uma aplicação Android como de interface gráfica entre a API e o usuário final, bem como um estudo de caso na cidade do Rio de Janeiro focado na modalidade de transporte público ônibus.

1.3 Organização do texto

O presente trabalho está estruturado em capítulos e, além desta introdução, será desenvolvido da seguinte forma:

- Capítulo II: Apresentamos a fundamentação teórica para a aplicação aqui proposta. Revisamos os conceitos de um processo de ETL - *Extract, Transform, Load*, sobre Web API e a arquitetura REST, além de apresentarmos algumas aplicações de apoio à mobilidade urbana já existentes e que nos baseamos para o desenvolvimento da Ikuzo API.

- Capítulo III: Discutimos toda a construção da aplicação desenvolvida neste trabalho, que denominamos Ikuzo API. Passaremos pela arquitetura projetada, a modelagem dos dados e as definições de implementação da aplicação.

- Capítulo IV: Mostramos o estudo de caso de uso da Ikuzo API utilizada na cidade do Rio de Janeiro com o desenvolvimento da aplicação Ônibus Carioca, para plataforma Android, e os resultados do estudo.

- Capítulo V: Reunimos as considerações finais, bem como as contribuições da pesquisa e estudos e desenvolvimentos futuros do tema.

2. Fundamentação Teórica

Neste capítulo são apresentados os componentes fundamentais para o estudo proposto, sobre os conceitos de um processo de ETL, continuaremos a discussão com Web APIs e sua a arquitetura REST, e por fim falaremos sobre aplicações disponíveis relacionadas com o tema.

2.1 ETL

Extract, Transform, Load (ETL) trata-se de um processo de carga de dados utilizando integrações entre sistemas, onde o dado é extraído de uma ou múltiplas fontes de dados, passa por uma sequência de transformações (ou mapeamentos) através de uma lógica ou engrenagem *ad hoc* e por fim são cadastrados em um banco de dados (ou *Data Warehouse*), seja buscando a limpeza ou enriquecimento dos dados como no trabalho de [4], ou mesmo para a união de informação de domínios heterogêneos. A Figura 2.1 ilustra o processo de ETL.

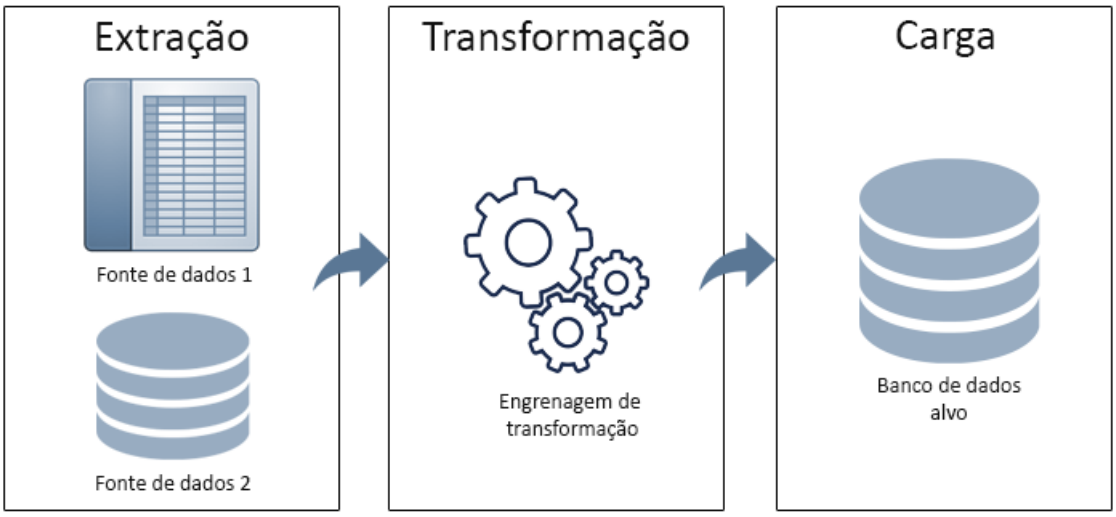

Figura 2.1: O processo de ETL

Em [6], são discutidas as seis tarefas que comumente compõem um processo de ETL:

- A seleção das fontes de dados para a extração;
- A transformação dos dados das fontes: Após os dados serem extraídos de suas fontes, eles podem ser transformados. Alguns procedimentos comuns são a filtragem dos dados, conversão de valores, cálculo de valores derivados, geração de números sequenciais para chaves auxiliares, entre outros;
- A junção de fontes de dados: Diferentes fontes podem ser unidas em busca de realizarem a carga de dados em um único alvo;
- A seleção do alvo a receber a carga: Um ou mais alvos serão escolhidos para receber a carga de dados;
- O mapeamento de atributos: Os atributos da base de dados fonte serão mapeados aos seus correspondentes no alvo;
- A carga de dados: O alvo será carregado com os dados agora transformados;

2.2 Web API

Web API (*Application Programming Interface*) é o mecanismo chave da interconectividade entre serviços de software pela internet [7]. Esta expõe parte de suas funcionalidades internas para qualquer usuário ou aplicação externa através de um contrato, fazendo-se possível o compartilhamento de dados e informações sem que haja a necessidade de domínio ou gerência sobre a informação a ser consumida.

Web APIs podem seguir diversos protocolos ou arquiteturas, como por exemplo, o REST, que entraremos em mais detalhes a seguir.

2.2.1 Arquitetura REST

Também chamada de RESTFul, alcunhada por [8], a Transferência de Estado Representacional (*Representational State Transfer*), é uma arquitetura baseada em HTTP que descreve como recursos da rede (ou aplicações web) devem ser definidos e endereçados, fazendo com que solicitantes tenham acesso a representações textuais de recursos da internet.

O REST, após décadas de sua concepção, ainda é uma das mais importantes tecnologias web disponíveis, podendo ser visto como o modelo de arquitetura padrão para a maioria das aplicações que acessam a internet [9].

No trabalho de [10] vemos alguns dos principais conceitos envolvidos na arquitetura REST:

- Recursos são referenciados e identificados na forma de URIs;
- Recursos podem ter diferentes representações (ou formatos), como por exemplo, HTML, XML, JSON ou texto simples. As representações serão enviadas no corpo da mensagem;
- Predefine métodos (ou verbos) HTTP que irão agir sobre os recursos: GET para obtenção de uma representação de um recurso; PUT para atualização do conteúdo de uma URI; DELETE para remoção de um recurso e POST para modificação ou adição de informações;
- Respostas devem conter cabeçalhos (*headers*), um código de status HTTP [1] e, possivelmente, um corpo (*payload*);

[1] Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content: https://tools.ietf.org/html/rfc7231

2.3 Aplicações relacionadas de apoio à mobilidade urbana

Já existem diversas aplicações de apoio à mobilidade urbana disponíveis para a população, seja em abrangência local ou global. Para embasar nossos estudos, selecionamos quatro dessas aplicações e analisamos suas características para o desenvolvimento da aplicação a ser proposta. Foram estudados o Google Maps, o aplicativo Moovit, Citymapper e a aplicação web Rio.bus.

2.3.1 Google Maps

Com mais de cinco bilhões de downloads na loja de aplicativos Google Play, é atualmente a aplicação de mapas com o maior número de usuários [2], o Google Maps fornece toda uma gama em abrangência global não somente no contexto de mobilidade urbana, mas sobre mapas em geral.

O Google Maps possui funcionalidades como pesquisa e escolha de melhores de rotas de maneira intermodal (carros, trens, ônibus, carros particulares, caminhada a pé, entre outros), além de um auxiliar para a navegação enquanto em rota com possibilidade de recálculo da rota a qualquer momento.

No contexto de transporte público, o Google Maps possui uma base de dados onde possui horários sobre qual o modal passaria em determinado local e realiza seus cálculos de rotas em cima destes valores. Os horários são informados em tempo real se assim disponível ou a partir de uma base de dados com horários fixos.

O Google Maps possui uma API na qual disponibiliza grande parte dos seus recursos, sendo que outras aplicações podem utilizá-los para novas implementações de sistemas de mapas ou mobilidade urbana, bastando que haja uma chave chamada de Api Key para a integração. A Api Key pode ser obtida de forma gratuita ou paga, caso haja necessidade de um alto número de requisições.

[2] Top 20 Most Used (and downloaded) Mobile Apps: https://blog.sagipl.com/most-used-apps/

2.3.2 Moovit

Com mais de 50 milhões de downloads na loja de aplicativos Google Play, o Moovit é uma popular aplicação de apoio à mobilidade urbana, fornecendo rotas e horários para modais de transporte público em abrangência global.

Possui funcionalidades como horário de chegada de modais de transporte, alertas de trânsito e relatórios de usuários. Os horários são definidos através de alertas em tempo real ou em uma base de dados já pré-cadastrada.

O Moovit não possui uma API gratuita aberta para obtenção de dados por sistemas externos, mas disponibiliza widgets (uma pequena aplicação de código-fonte fechado) sendo possível anexar a aplicações terceiras.

2.3.3 Citymapper

Com mais de cinco milhões de downloads na loja de aplicativos Google Play, Citymapper é uma aplicação de apoio à mobilidade urbana disponível para dezenas das maiores cidades do mundo e possui informações intermodais de transportes públicos e privados.

O Citymapper fornece informações de cálculo de tempo de percurso, rotas a serem tomadas e modais a serem tomadas, é tido como um dos melhores aplicativos para transportes públicos devido a sua grande quantidade e qualidade de dados fornecidos a seus usuários [3].

O Citymapper disponibiliza uma API para integração e obtenção de seus dados por aplicações externas através de uma chave de API que pode ser obtida após um simples cadastro, porém possui limitações sobre a quantidade de requisições que podem não ser suficientes para atender uma aplicação terceira de não tão pequeno porte.

[3] Innovative Urban Transportation Apps: https://meetingoftheminds.org/12-innovative-urban-transportation-apps-4708

2.3.4 Rio.bus

Rio.bus trata-se de um sistema colaborativo de monitoramento de ônibus em tempo real que utiliza uma API de dados abertos fornecida pela prefeitura do estado do Rio de Janeiro com o objetivo de ajudar o usuário a se deslocar pela cidade.

O Rio.bus possui funcionalidades, além da posição de ônibus em tempo real, informações de itinerário de linhas de ônibus na cidade do Rio de Janeiro.

O Rio.bus tem seu código-fonte aberto e possui uma API totalmente aberta para a obtenção das informações que disponibiliza em seu sistema.

2.4 Considerações finais

Neste capítulo vimos os conceitos que servem de fundamentos para a aplicação de apoio à mobilidade urbana proposta nesse trabalho. Vimos os conceitos da técnica de *Extract, Transform, Load* (ETL) e Web API.

Quanto às aplicações já existentes de apoio à mobilidade urbana, vimos quatro diferentes, porém todas com o mesmo objetivo de apoiar a complexa questão da mobilidade urbana. Conseguimos levantar pontos fortes de cada aplicação, como a abrangência global do Google Maps e do Moovit, a quantidade de informação fornecida pela Citymapper e a integração focada em sistemas de prefeituras locais para obtenção de dados como na API da Rio.bus. No capítulo seguinte iremos, portanto, propor uma aplicação que visa unir os melhores pontos das aplicações citadas.

3. A aplicação Ikuzo API

Neste capítulo será proposto uma solução de aplicação de apoio à mobilidade urbana, que demos o nome de Ikuzo API, visando à informação sobre transportes públicos, como informações georreferenciais. O nome tem a origem na expressão japonesa *Ikuzo* (行くぞ) que, em uma tradução livre para o português, significa "vamos lá", ilustrando assim a relação da aplicação com mobilidade.

Na primeira parte será abordado a arquitetura para a aplicação e suas modelagens, após será debatido sobre a implementação aplicada com todas as suas funcionalidades e fluxos de dados.

3.1 Arquitetura

Para a arquitetura geral foi projetado uma aplicação que seja independente de local para uso, de linguagens de programação ou sistemas operacionais para acesso aos recursos disponíveis, sendo possível que clientes que consumam os dados consigam fazê-lo seja através de um site na web ou dispositivos e aplicações *mobile*. Seguindo os conceitos de ETL, A aplicação também deve estar conectada à servidores de informações sobre dados de transporte público, principalmente dados georreferenciais e de rotas, além de estar integrada a servidores de geolocalização para validação, tratamento e aceite dos dados obtidos. A Figura 3.1 mostra essa arquitetura geral proposta.

3.2 Modelagem de dados

Será demostrado nessa seção como os dados da aplicação foram modelados para todo seu fluxo e persistência.

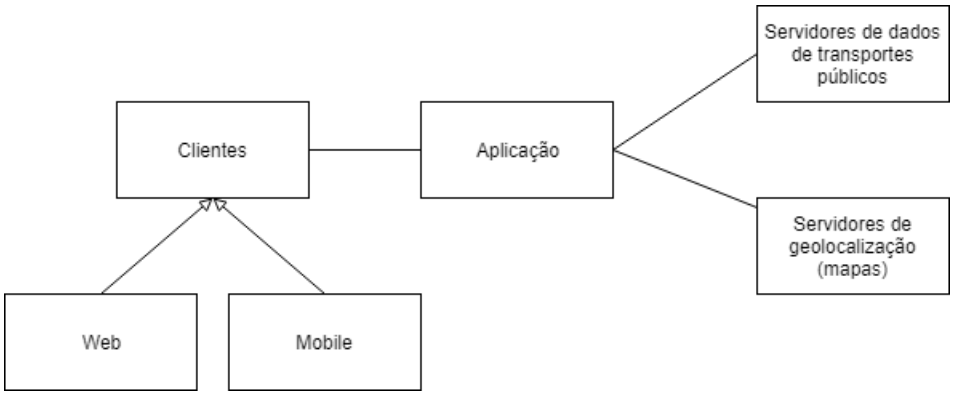

Figura 3.1: Arquitetura geral proposta

3.2.1 Classes de domínio

Foram definidas quatro classes de domínio principais para a aplicação, buscando a abstração dos dados a serem mantidos pela aplicação, tendo cada uma delas uma responsabilidade de escopo bem definida. São elas *Line, Modal, Gps* e *Itinerary*. a Figura 3.2 mostra a modelagem de classe proposta segundo modelagem UML.

- ***Modal*** - Representa um veículo de transporte público, deve possuir um identificador único para ser indexado.
- ***Line*** - Responsável por manter os dados referentes às linhas de modais. Pode ser visto como um agrupador de modais. Durante o trabalho, essa classe também será chamada de "linha".
- ***Itinerary*** - Mantém informações georreferenciais sobre o percurso e rotas sobre quais as linhas de modais percorrem durante todo o trajeto. Cada registro de Itinerary deve ser um ponto sobre o qual uma linha trafega, portanto, o conjunto de registros Itinerary de uma mesma linha define o percurso base completo da linha. Ao decorrer do trabalho, essa classe também poderá ser chamada de "ponto de itinerário", ou simplesmente "itinerário".
- ***Gps*** - Mantém a informação mais recente do geoposicionamento de um modal.
- ***Tag*** - Informação adjetiva a linha, para informações como linhas turísticas, linhas de trajeto rápido, entre outros.

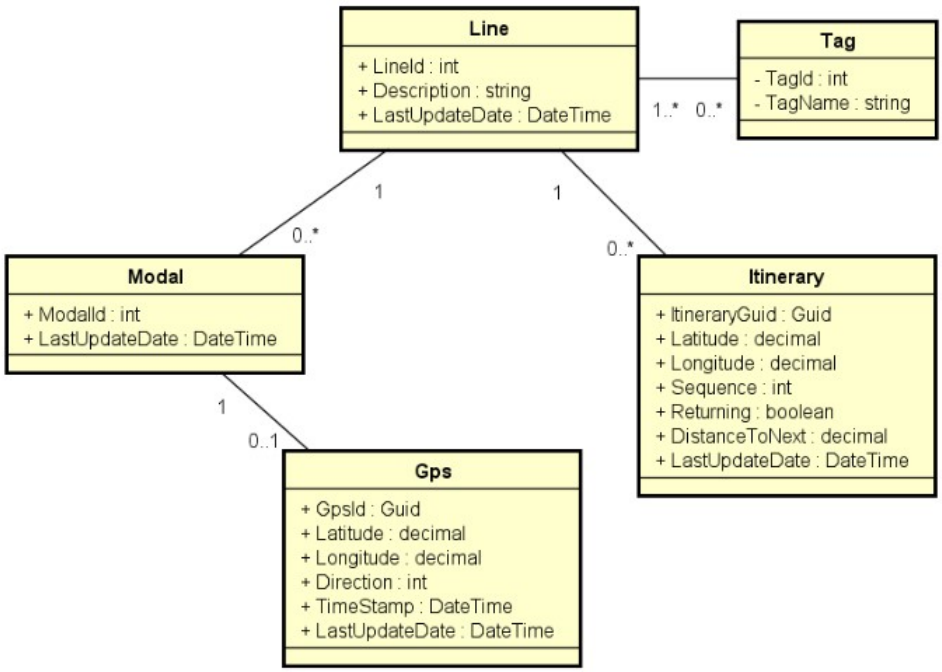

Figura 3.2: Modelagem de classes de domínio da Ikuzo API

3.2.2 Banco de dados

Para a persistência dos dados da aplicação, foi pensado em um modelo relacional de banco de dados que suporta as operações necessárias que serão realizadas sobre as classes de domínio. Cada esquema (ou tabela) e seus campos são descritos a seguir.

- ***Line***
  - *LineId*: Identificador único e interno da linha, representado por um número inteiro.
  - *Description*: Breve texto com informação da linha.
  - *LastUpdateDate*: Data e hora da última atualização do registro.
- ***Modal***
  - *ModalId*: Identificador único e interno do modal, representado por um número inteiro.
  - *LastUpdateDate*: Data e hora da última atualização do registro.
- ***Tag***
  - *TagId*: Identificador único e interno da tag, representado por um número inteiro.

  - *Name*: Nome de determinada característica que será associada a linha.

- ***LineTag*** - Tabela intermediária que gerada através da relação entre *Line* e *Tag*. Possui o identificador de cada tabela, bem como um identificador exclusivo da relação.

- ***Itinerary***

  - *ItineraryId*: Identificador único e interno do registro de itinerário, representado por um Identificador Único Universal (UUID).
  - *Latitude*: Coordenada geográfica referente à latitude do registro de itinerário, representada por um número com até 6 casas decimais.
  - *Longitude*: Coordenada geográfica referente à longitude do registro de itinerário, representada por um número com até 6 casas decimais.
  - *Sequence*: Número sequencial representativo da posição do ponto de itinerário quanto ao conjunto total de pontos do itinerário.
  - *Returning*: Informação booleana se o registro de itinerário se trata de um ponto da ida ao destino ou ao retorno à origem.
  - *DistanceToNext*: Distância em metros do ponto de itinerário ao próximo da sequência.
  - *LastUpdateDate*: Data e hora da última atualização do registro.

- ***Gps***

  - *GpsId*: Identificador único e interno do registro de GPS, representado por um Identificador Único Universal (UUID).
  - *Latitude*: Coordenada geográfica referente à latitude do registro de GPS, representada por um número com até 6 casas decimais.
  - *Longitude*: Coordenada geográfica referente à longitude do registro de GPS, representada por um número com até 6 casas decimais.
  - *Direction*: Número inteiro representando graus à qual posição o modal estava apontado durante a última extração de posicionamento global.
  - *TimeStamp*: Data e hora da última extração de posicionamento global.
  - *LastUpdateDate*: Data e hora da última atualização do registro.

- ***GpsHistory*** - Tabela réplica dos dados da tabela *Gps* que persiste todo o histórico dos dados georreferenciais, diferentemente da tabela *Gps*, que possuirá somente os dados da última extração.

A Figura 3.3 mostra o modelo relacional de banco de dados com os esquemas descritos acima bem como seus relacionamentos.

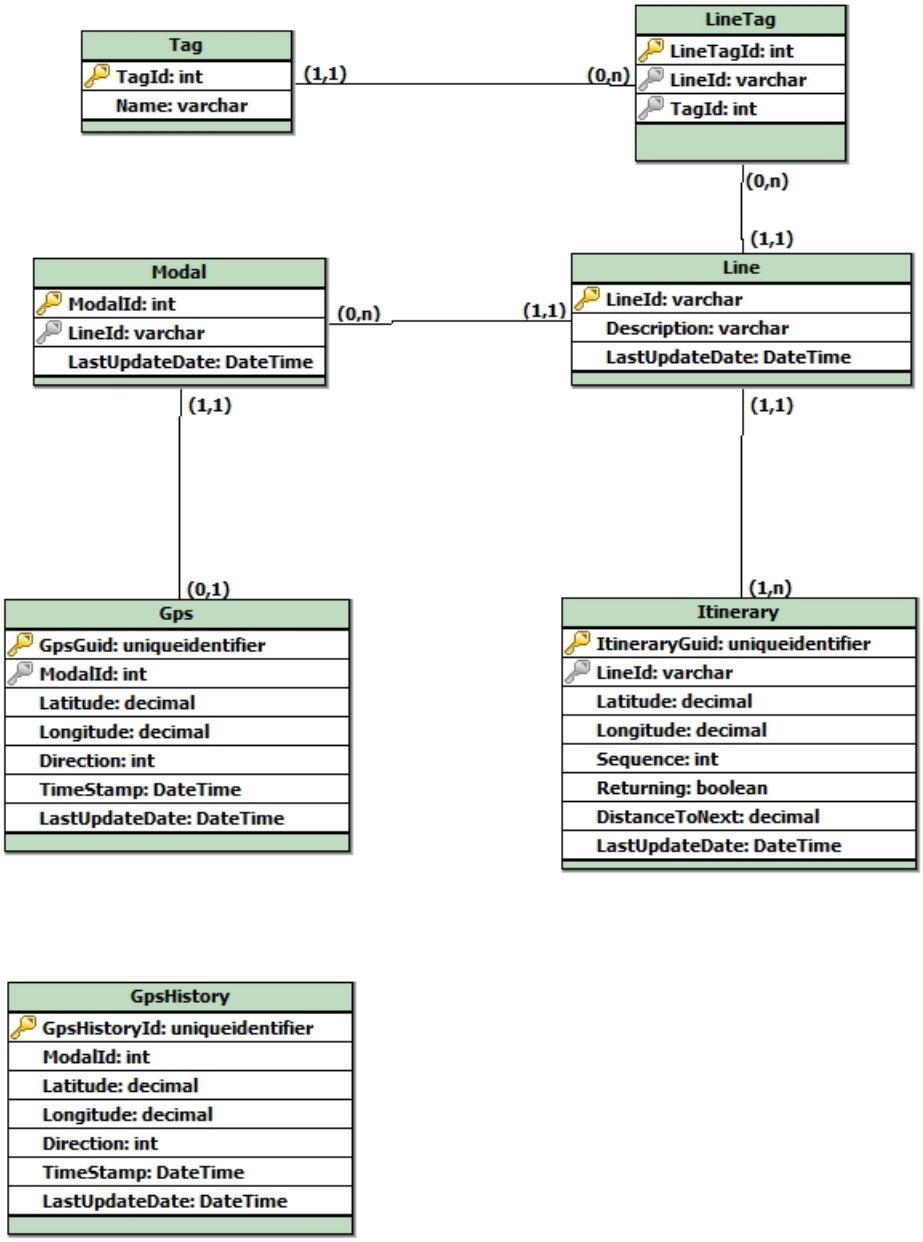

Figura 3.3: Modelo relacional

3.3 Implementação da aplicação

A Ikuzo API funciona sobre o protocolo HTTP, utilizando a arquitetura cliente-servidor. O corpo das respostas às requisições é dado no formato JSON, possibilitando a qualquer tecnologia que também funcione sobre o protocolo e interprete JSON estabelecer uma integração. Foi desenvolvida sobre a linguagem de programação C#, utilizando o .NET Framework e tendo a persistência no banco de dados Microsoft SQL Server.

3.3.1 Arquitetura de desenvolvimento

A Ikuzo API foi escrita seguindo o padrão de arquitetura Modelo-Visão-Controladora (*Model-View-Controller* – MVC)[1] e para melhor abstração e divisão de responsabilidades a dividimos em quatro camadas:

- ***Api*** - Camada onde estão localizados todos os controladores e definições da API bem como as rotas de acesso para cada recurso.
- ***App*** - Camada que serve como orquestrador das requisições da camada Api levando-as as classes da camada de domínio e infraestrutura.
- ***Domain*** - Camada onde se encontram todas as classes de domínio e suas regras de negócio.
- ***Infra*** - Camada onde se encontram todas as configurações de conexão a banco de dados, repositórios internos e externos, configurações e mapeamentos de interface de classes.

A Figura 3.4 mostra cada uma das camadas citadas acima juntamente com sua inter-dependência.

[1] Padrão de arquitetura MVC: https://dotnet.microsoft.com/apps/aspnet/mvc

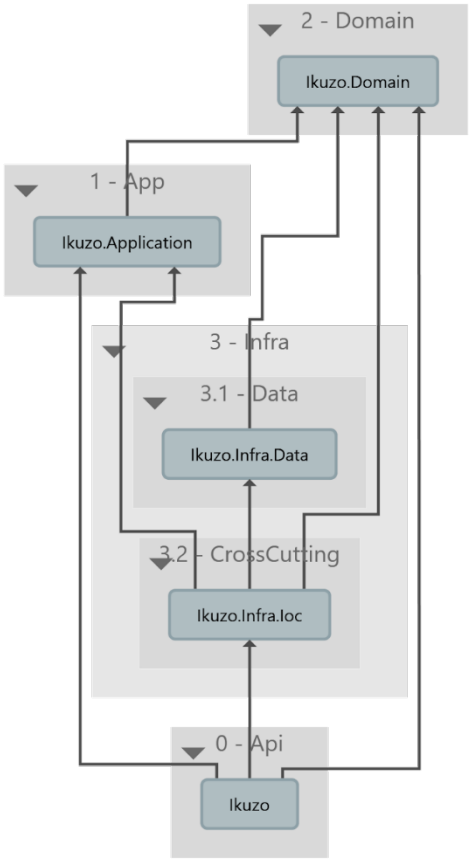

Figura 3.4: Arquitetura de camadas na Ikuzo API e sua interdependência

3.3.2 Conexão ao banco de dados

A Ikuzo API se mantém sua conexão com seu banco de dados SQL Server através da biblioteca .NET Dapper [2]. A biblioteca foi escolhida levando em consideração testes e estudos como o de [11] que comprovam a maior eficiência da biblioteca Dapper sobre outras, como por exemplo, o Entity Framework, da Microsoft.

3.3.3 Segurança e autenticação

Para acessar os recursos de atualização de dados, é necessário adicionar no cabeçalho da requisição HTTP um campo denominado AccessKey que possui um valor em formato de Identificador Único Universal (UUID) [12] que deve ser igual ao registrado no servidor onde a Ikuzo API está em execução. A não informação do campo AccessKey no cabeçalho da requisição ou a informação incorreta do mesmo ocasiona o código HTTP 401 - Unauthorized [13], sendo descartada a requisição.

[2]Dapper: https://github.com/StackExchange/Dapper

3.3.4 Definição de recursos da aplicação

Para requisitar qualquer recurso da Ikuzo API, foi definido um padrão de URN – *Uniform Resource Name* [14] onde a primeira parte trata a definição da aplicação, seguindo a versão, o recurso solicitado e por fim os parâmetros de filtragem. Cada rota disponível é descrita conforme a Tabela 3.1:

Rota	**Verbo HTTP**	**Parâmetros**	**Descrição**
/api/v1/lines	GET		Obtém todas as linhas cadastradas.
/api/v1/lines/{lineId}	GET	lineId : string	Obtém informações da linha requisitada.
/api/v1/lines/{lineId}/itinerary	GET	lineId : string	Obtém todo o itinerário da linha requisitada.
/api/v1/lines/local	GET	lat : decimal lon : decimal distance : decimal tags : string	Obtém lista com todas as linhas que possuem itinerário em um determinado ponto, no alcance máximo determinado pelo parâmetro distance.
/api/v1/modals	GET		Obtém todos os modais cadastrados.
/api/v1/modals/{modalId}	GET	modalId : string	Obtém informações do modal requisitado.
/api/v1/modals/nearby	GET	lat : decimal lon : decimal distance : decimal line : string	Obtém modais próximos a um determinado ponto, no alcance máximo determinado pelo parâmetro distance.
/api/v1/itineraries/route	GET	latOri : decimal lonOri : decimal latDest : decimal lonDest : decimal tags : string	Obtém lista de linhas que possuem itinerários próximos a um ponto de origem e um de destino, com sua respectiva distância de percurso.
/api/v1/crawler/gps	POST		Atualiza informações de GPS.
/api/v1/crawler/lines	POST		Atualiza informações de linhas.
/api/v1/crawler/modals	POST		Atualiza informações de modais.
/api/v1/crawler/itineraries	POST		Atualiza informações de itinerário.

Tabela 3.1: Definição de recursos da Ikuzo API

3.3.5 Modelagem de sequência

Nesta seção será demonstrado todo o fluxo de dados entre os componentes da aplicação para todas as funcionalidades disponíveis. As funcionalidades serão divididas por cada URN no qual o recurso é acessado. As modelagens sempre seguirão o padrão adotado de terem seu fluxo iniciado nas classes controladoras, repassando para as classes de aplicação que tratam a requisição chamando classes de serviço e classes de repositório que se comunicam com o banco de dados. O fluxo sempre se encerrará na classe controladora que iniciou o fluxo com o resultado da operação.

Temos as classes Crawler, responsáveis por gerenciar as requisições de atualização de dados da aplicação, classes Line, responsáveis por manter informações de linhas, classes Modal que mantém modais, Itinerary responsáveis por itinerários e Gps responsáveis pelos registros de GPS. A classe ExternalRepository é a classe que faz a ligação entre a Ikuzo API e serviços/sistemas externos para obtenção de dados.

- **/api/v1/crawler/lines**

Esse recurso tem por objetivo a atualização das linhas cadastradas no banco de dados da aplicação.

A requisição tem início na classe controladora CrawlerController que invoca o recurso *SyncLines()* da classe CrawlerApp, que por sua vez chama a classe responsável pela integração com sistemas externos para obter todas as linhas. Essas informações serão retornadas até a CrawlerApp para fazer a verificação das linhas e editá-las ou criá-las a partir de funções disponíveis na classe de serviço LineService que chama métodos de acesso ao banco de dados disponiveis na classe de repositório LineRepository. A Figura 3.5 mostra a modelagem de sequência para atualização do recurso da Ikuzo API Lines.

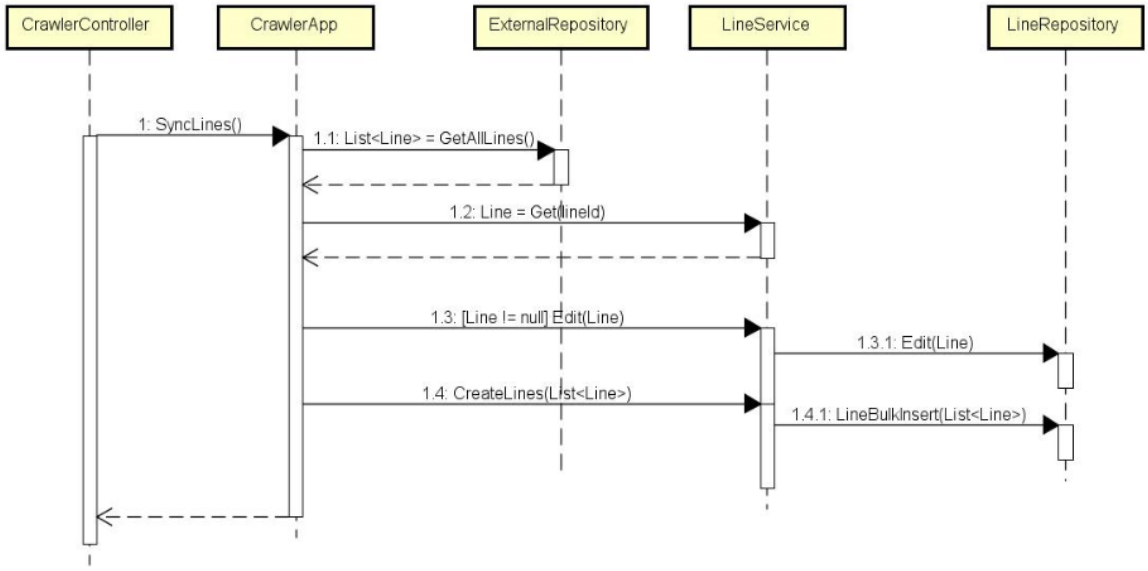

Figura 3.5: Modelagem de sequência para atualização de linhas

- **/api/v1/crawler/itineraries**

Esse recurso tem por objetivo a atualização dos segmentos de itinerários cadastrados no banco de dados da aplicação.

O fluxo se inicia na controladora CrawlerController chamando o recurso *SyncItineraries()* da classe de aplicação CrawlerApp, esta irá obter todas as linhas pela classe LineService e todos os itinerários cadastradas no recurso disponível de repositórios externos. Para cada ponto de itinerário da linha, será calculado a distância entre o ponto e o próximo da sequência determinado pelo parâmetro *Sequence* através da fórmula de Haversine (descrita a seguir), procedimento adotado para obtermos a distância total de percurso de um itinerário de uma linha. O fluxo continua por remover informações de itinerários antigos e cadastrar novos pontos de itinerários em lote através da classe de serviço ItineraryService e da classe de banco de dados ItineraryRepostiry. A Figura 3.6 mostra a modelagem de sequência para atualização do recurso da Ikuzo API Itineraries.

- **Fórmula de Haversine**

A fórmula de Haversine (3.1) é a equação matemática utilizada principalmente em sistemas de navegação para obtenção da distância entre dois pontos em uma esfera dada uma latitude e uma longitude, utilizando relações trigonométricas.

$$d = 2r \arcsin\left(\sqrt{\sin^2\left(\frac{\theta_2 - \theta_1}{2}\right) + \cos(\theta_1)\cos(\theta_2)\sin^2\left(\frac{\alpha_2 - \alpha_1}{2}\right)}\right) \quad (3.1)$$

Onde:

r: Raio
θ_1: Latitude origem
θ_2: Latitude destino
α_1: Longitude origem
α_2: Longitude destino

A Ikuzo API possui implementações da fórmula de Haversine tanto em seu código, utilizando a biblioteca C# GeoCoordinate [3] quanto em seu banco de dados, através da função SQL proposta em [15].

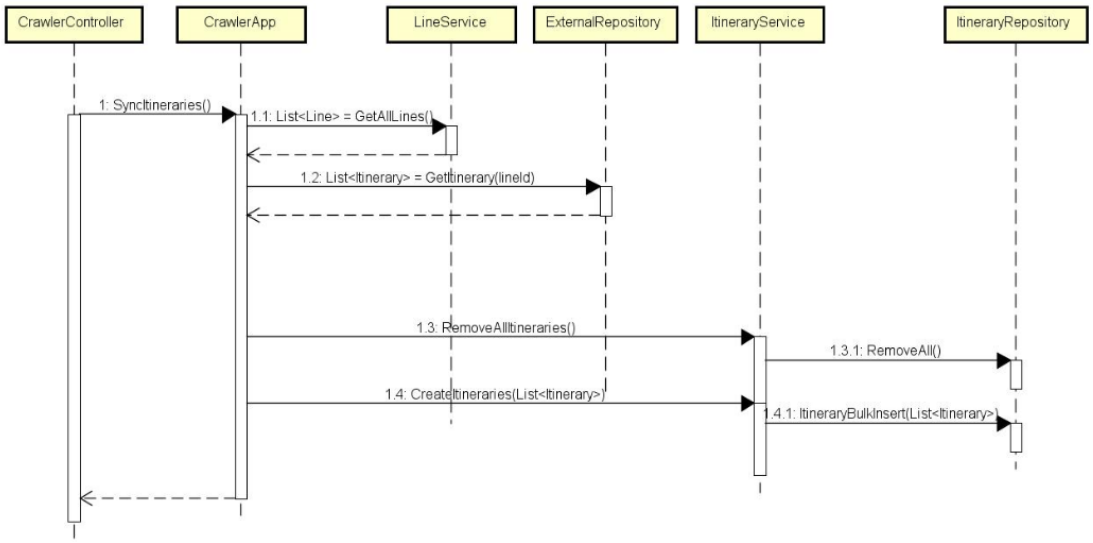

Figura 3.6: Modelagem de sequência para atualização de itinerários

[3] GeoCoordinate Class: https://docs.microsoft.com/pt-br/dotnet/api/system.device.location.geocoordinate

- **/api/v1/crawler/modals**

Esse recurso tem por objetivo a atualização dos modais cadastrados no banco de dados da aplicação.

O fluxo de dados começa na classe CrawlerController que irá invocar a classe CrawlerApp para orquestrar e executar a requisição: Primeiro serão obtidas todas as linhas de um repositório externo de dados e para cada linha obtida, iremos fazer uma consulta se esta existe no banco de dados, através da classe de serviço LineService, existindo, iremos buscar todos os modais associados a essa linha também em recursos disponíveis de repositórios fora da aplicação. Com todas as modais obtidas de uma linha, faremos a remoção de todos os registros de modais da linha que possuíamos cadastrado com uma chamada a classe ModalService e a classe ModalRepository para efetivamente se conectar e remover do banco de dados. Após remoção concluída, iremos cadastrar todos os modais obtidos no passo anterior. A Figura 3.7 mostra a modelagem de sequência para atualização do recurso modals.

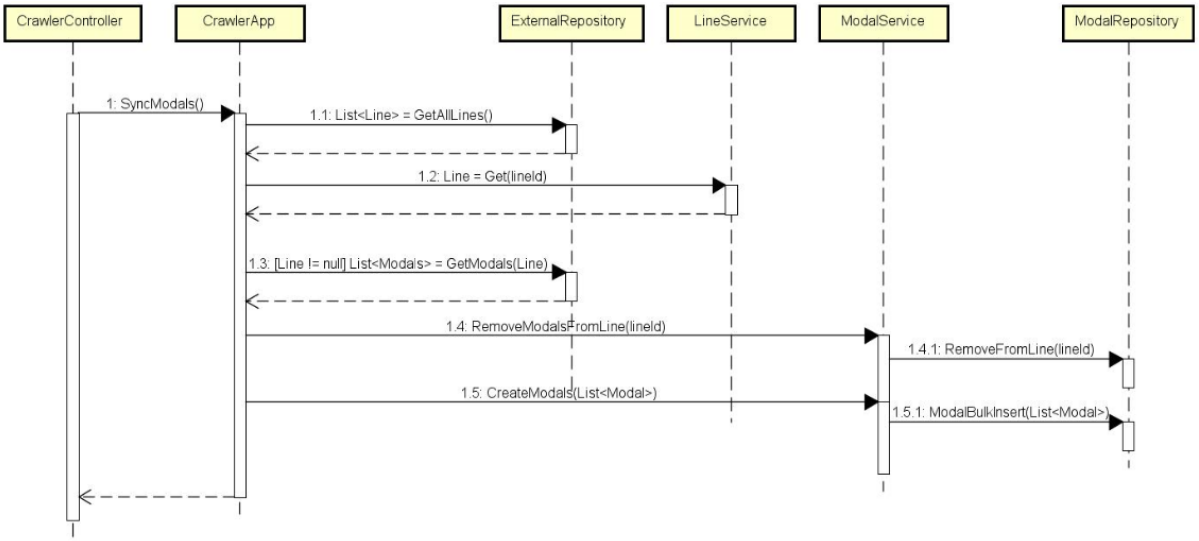

Figura 3.7: Modelagem de sequência para atualização de modais

- **/api/v1/crawler/gps**

Esse recurso tem por objetivo a atualização das linhas cadastradas no banco de dados da aplicação.

Iniciando na classe controladora CrawlerController e repassada para a classe CrawlerApp, iremos consultar as informações de GPS disponíveis no repositório externo de dados. Caso haja dados na consulta, segue por chamar o método da classe GpsService *RemoveAllGps()* responsável por invocar o método *RemoveAll()* da classe GpsRepository que tem a finalidade de remover do banco de dados todos os registros desatualizados de GPS. Durante o processo de exclusão de dados, estes serão armazenados em uma tabela de histórico, para que possam ser feitos estudos posteriores. No passo seguinte, haverá o cadastro das informações atualizadas na tabela Gps. A Figura 3.8 mostra a modelagem de sequência para atualização do recurso Gps.

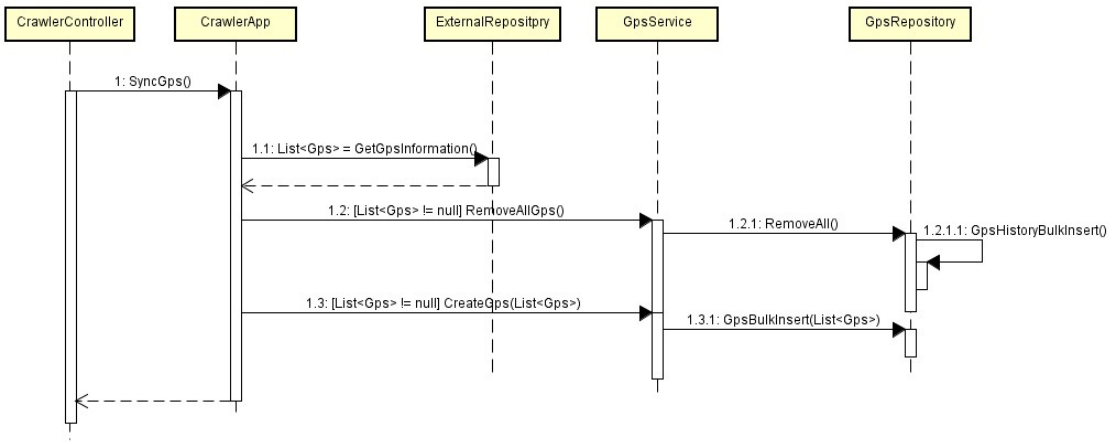

Figura 3.8: Modelagem de sequência para atualização de GPS

- **/api/v1/itineraries/route**

Esse recurso tem por objetivo a obtenção de linhas que possuam itinerário próximos a um par de longitude e longitude tanto de origem quanto de destino.

Com início na ItineraryController e informado o par latitude e longitude, tanto de origem quanto destino, e uma distância máxima (em metros) do raio de distância aceitável dos pontos, já na classe ItineraryApp, iremos obter através do recurso disponível na classe ItineraryService todas as linhas que possuem o itinerário que passam por ambos os pontos (origem e destino) informados no início da requisição, a consulta ao banco de dados será feita na classe ItineraryRepository. Para cada linha recuperada, é calculada a distância total percorrida para aquele ponto de itinerário, do ponto de origem ao ponto de destino. A Figura 3.9 mostra a modelagem de sequência para consultas no recurso route.

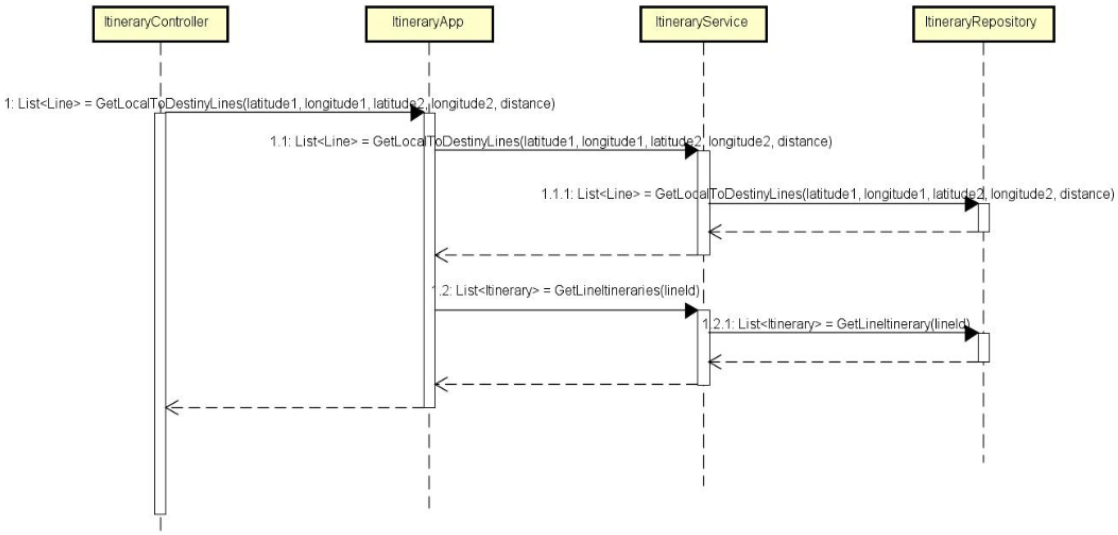

Figura 3.9: Modelagem de sequência para obtenção de rota

- **/api/v1/lines**

Esse recurso tem por objetivo obtenção de todas as linhas cadastradas no banco de dados da aplicação.

O fluxo, iniciando na classe controladora LineController e orquestrada pela LineApp, será a consulta e retorno de todas as linhas cadastradas previamente. A consulta será feita através do recurso disponível na classe LineService, acessando a LineRepository para consulta ao banco de dados. Caso não haja linhas cadastradas, será retornada uma lista vazia. A Figura 3.10 mostra a modelagem de sequência para consultas no recurso Lines.

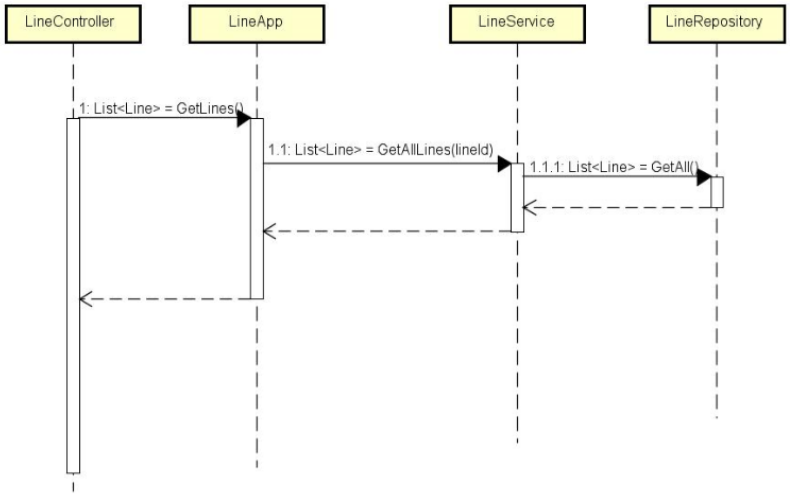

Figura 3.10: Modelagem de sequência para listagem de linhas

- **/api/v1/lines/{lineId}**

Esse recurso tem por objetivo a obtenção, se existir, de uma linha específica com um identificador informado no banco de dados da aplicação.

Com início na classe LineController, passado um identificador de linha, será repassado para a classe LineApp para que seja feito uma consulta no recurso para se obter detalhes da linha com identificador solicitado na classe LineService que irá solicitar a consulta ao banco de dados para a classe LineRepository. Serão retornadas informações específicas da linha, caso ela seja encontrada. A Figura 3.11 mostra a modelagem de sequência para consultas dado o identificador de linha lineId no recurso Lines.

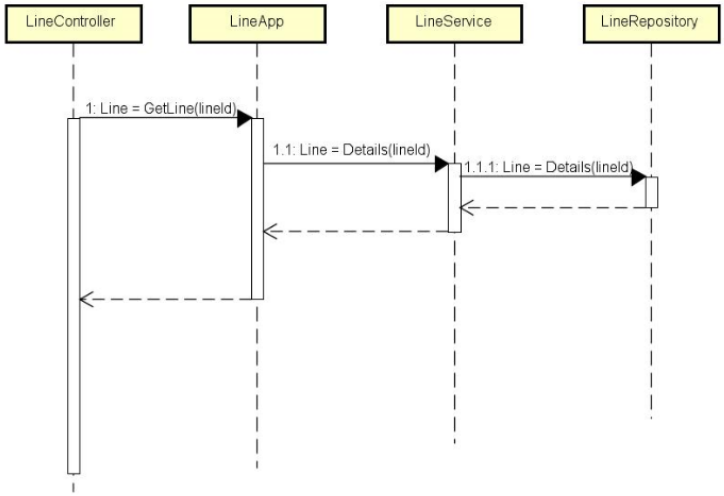

Figura 3.11: Modelagem de sequência para detalhes de uma linha

- **/api/v1/lines/{lineId}/itinerary**

Esse recurso tem por objetivo a obtenção dos pontos de itinerário, formando o itinerário completo de uma linha de identificador informado cadastrada no banco de dados da aplicação.

Dado um identificador de linha, começando na classe LineController e continuando na LineApp, realizamos uma consulta às informações de itinerário dessa linha. Primeiramente, iremos certificar de que o identificador seja de uma linha cadastrada, através das classes LineService e LineRepostiory para busca no banco de dados. Existindo a linha, iremos invocar a classe ItineraryRepository para obtermos os pontos de itinerário da linha, montando seu itinerário completo. A Figura 3.12 mostra a modelagem de sequência para consultas no recurso itinerary, informado um identificador de linha.

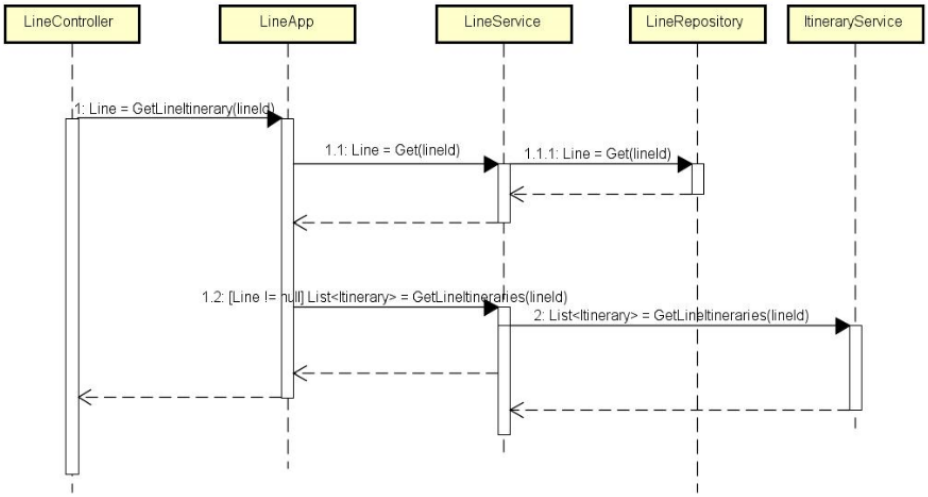

Figura 3.12: Modelagem de sequência para listagem de itinerários

- **/api/v1/lines/local**

Esse recurso tem por objetivo a obtenção de todas as linhas que possuam um ponto de itinerário próximo a um ponto geoespacial (latitude, longitude) informado, dessa forma, recuperando todas as linhas que passam próximas àquele local.

Tendo início na LineController e executada pela classe LineApp, a requisição informada com um par latitude e longitude, além de um parâmetro auxiliar de distância máxima em metros do ponto, será feita uma consulta na classe LineService de todas as linhas que possuem um ponto de itinerário próximo, no máximo, à distância solicitada. A classe LineService irá realizar a consulta às informações cadastradas de itinerário no banco de dados através da ItineraryRepository. Caso nenhum registro seja encontrado, será retornada uma lista vazia. A Figura 3.13 mostra a modelagem de sequência para consultas no recurso local.

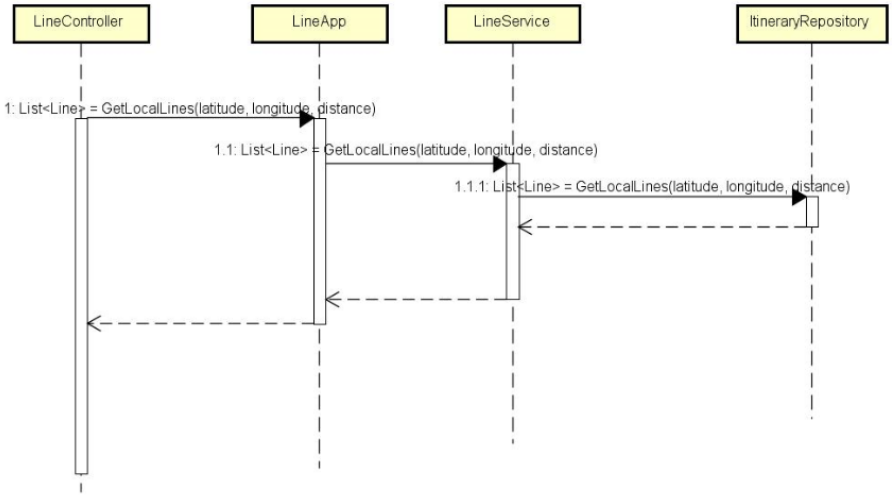

Figura 3.13: Modelagem de sequência para obtenção de linhas de um local

- **/api/v1/modals**

Esse recurso tem por objetivo a obtenção de todos os modais cadastrados no banco de dados da aplicação.

Com início na ModalController e enviado para a ModalApp, será feito uma consulta no recurso da classe ModalService sobre todas os modais cadastrados no banco de dados através da classe ModalRepository. Caso não haja modais, será retornada uma lista vazia. A Figura 3.14 mostra a modelagem de sequência para consultas no recurso modals.

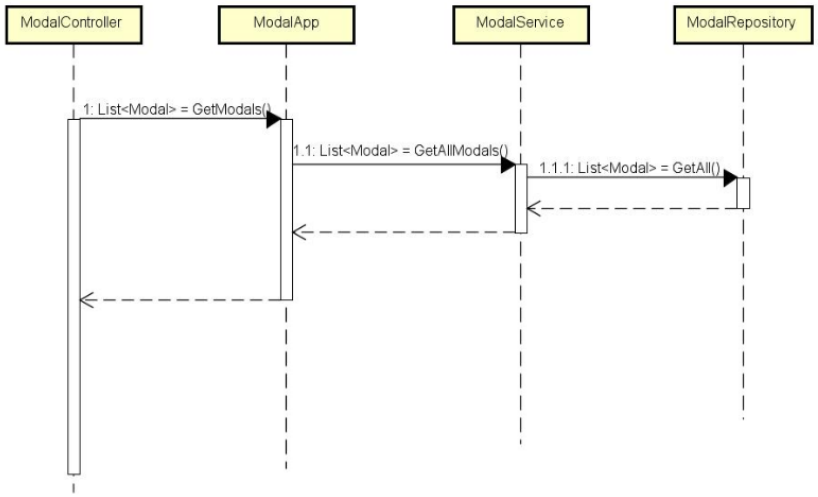

Figura 3.14: Modelagem de sequência para listagem de modais

- **/api/v1/modals/{modalId}**

Esse recurso tem por objetivo a obtenção, caso exista, das informações detalhadas de um modal com um identificador informado.

Partindo da ModalController e encaminhada para a ModalApp, O fluxo será uma consulta orientada a um identificador de modal requisitado. Caso esteja cadastrado, serão retornadas informações específicas do modal, juntamente com a informação atual de geoposicionamento do mesmo. Essa funcionalidade estará disponível na classe ModalService para obtenção das informações de modal e, caso modal exista, serão obtidas as informações de GPS na classe GpsService. A Figura 3.15 mostra a modelagem de sequência para consultas, dado identificador, no recurso modals.

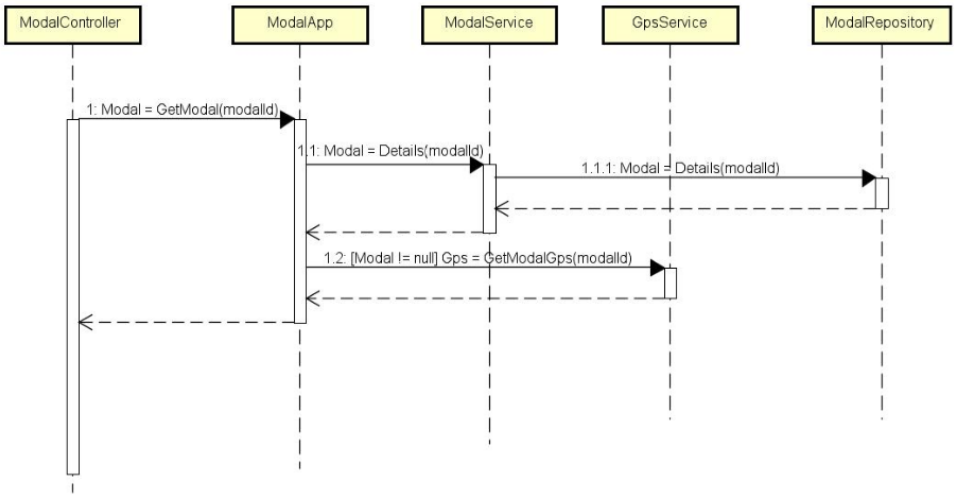

Figura 3.15: Modelagem de sequência para detalhes de um modal

- **/api/v1/modals/nearby**

Esse recurso tem por objetivo a obtenção dos modais próximos a um ponto geoespacial informado no momento da consulta.

O fluxo, iniciando na ModalController e repassada para a ModalApp, através do ponto georreferencial informado e a distância máxima de tolerância, será feito uma consulta através do método disponível na classe GpsService, que realiza a consulta ao banco de dados através da classe GpsRepository, os modais que estão, no momento, próximos ao ponto informado. A informação de distância é calculada através da fórmula de Harversine. Caso nenhum modal encontrado, será retornado uma lista vazia. A Figura 3.16 mostra a modelagem de sequência para consultas no recurso nearby.

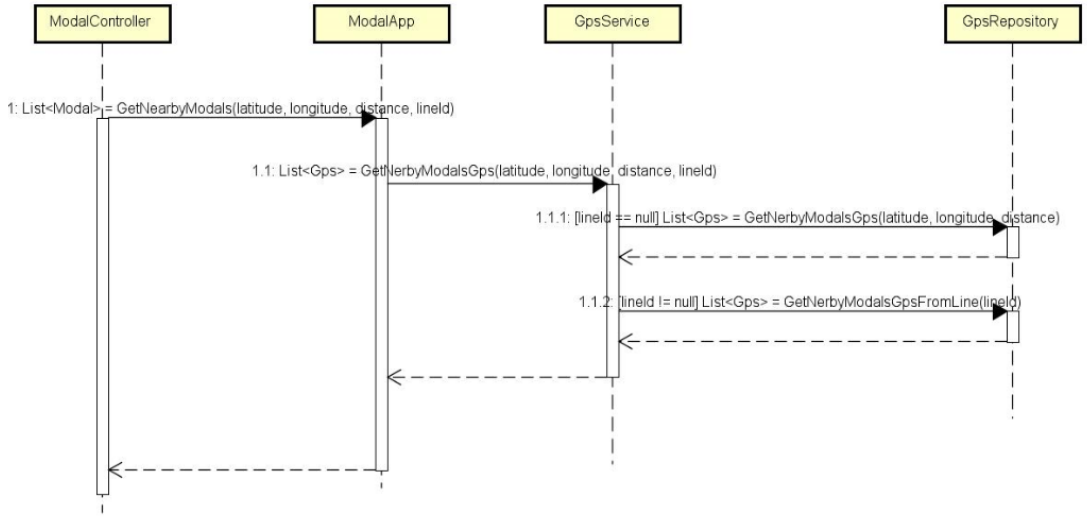

Figura 3.16: Modelagem de sequência para obtenção de modais próximos a uma latitude e longitude

3.4 Considerações finais

Neste capítulo vimos toda a modelagem de dados da aplicação, seus fluxos, definições e implementações. Todo o código fonte da Ikuzo API é aberto e pode ser encontrado em: https://github.com/bruno-sales/Ikuzo-api.

No capítulo seguinte, veremos um estudo de caso com o uso da aplicação proposta, utilizando-a no contexto de ônibus da cidade do Rio de Janeiro.

4. Estudo de caso

Neste capítulo será demonstrado o estudo de caso da solução de aplicação proposta com sua aplicabilidade no município do Rio de Janeiro, no contexto de transportes públicos no modal Ônibus.

Nas próximas seções serão apresentados os repositórios externos de dados nos quais a Ikuzo API foi integrada para obtenção de dados, como foi feita a atualização destes dados, a aplicação Android Ônibus Carioca desenvolvida para ser uma interface dos resultados dos recursos da Ikuzo API e por fim uma análise dos resultados obtidos.

4.1 Repositórios de dados

Para aplicação do estudo na cidade do Rio de Janeiro, a Ikuzo API foi integrada a dois repositórios de dados, o Data.rio e o Rio.Bus. Ambos os repositórios tratam o mesmo objeto comum e, portanto, a Ikuzo API os utiliza como contingência mútua para caso ocorra alguma indisponibilidade momentânea em algum dos mesmos. Seguindo os princípios de ETL, os dados são extraídos dos repositórios, mapeados para objetos de classe da Ikuzo API, calculados e salvos em seu banco de dados.

4.1.1 Data.Rio

Provido pela SIURB, o Sistema Municipal de Informações Urbanas, possui diversos repositórios de dados mantidos pela prefeitura do Rio de Janeiro, sendo um deles sobre transportes públicos.

A Ikuzo API foi integrada ao Data.Rio para consumir o recurso disponível de posição em tempo real dos ônibus e para itinerário das linhas de ônibus. A obtenção dos dados é feita através de uma API RESTFul, com o corpo da resposta da requisição no formato

comma separated value (csv) que é tratado na aplicação, convertendo-o para um objeto de classe e registrado no banco de dados.

4.1.2 Rio.bus

É uma aplicação de código-fonte aberto sobre informações de transporte público no Rio de Janeiro. A aplicação também é integrada no Data.rio e possui uma API aberta para consumo de recursos que trata e normaliza das informações obtidas no Data.rio. O corpo da resposta da requisição é dado no formato JSON.

A Ikuzo API utilizou os recursos de Linha, Ônibus e GPS disponibilizados pela API proxy disponibilizada pela Rio.Bus.

A Figura 4.1 demonstra como ficam arquitetadas as integrações entre as aplicações no contexto aplicado.

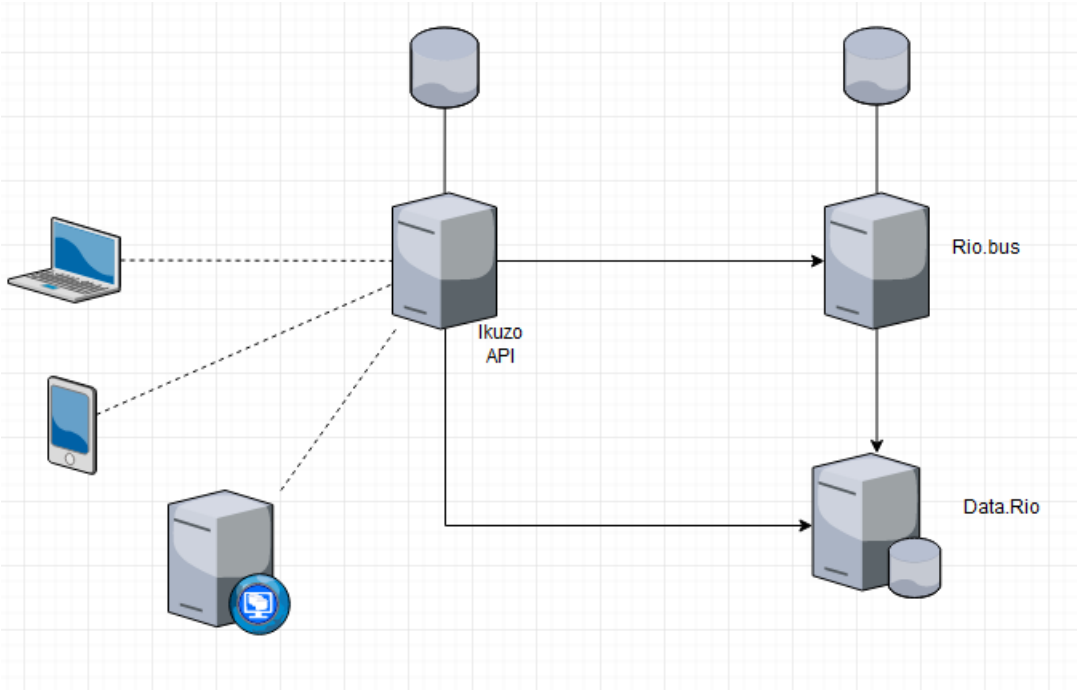

Figura 4.1: Arquitetura entre as aplicações

4.2 Atualização de dados

Os dados da aplicação, principalmente os referentes a GPS, precisam estar constantemente sendo atualizados e para isso foi escrito simples programa que a cada minuto realiza uma requisição ao recurso disponível e assim atualizando os dados da tabela do banco de dados interno da Ikuzo API. As informações de GPS, Linha, Modal e Itinerário

são atualizados ao consumir os recursos de /api/v1/crawler, agora com a busca de dados nos repositórios externos do Data.rio e do Rio.Bus.

Para que fossem feitas as atualizações com uma periodicidade definida e de maneira automática, foi desenvolvido um simples programa, na linguagem C#, denominado Ikuzo Bot. Seu código-fonte instancia um objeto temporizador que a cada minuto executa uma função *callback* [1] que faz a chamada por REST ao recurso. A Figura 4.2 mostra o código para a definição da periodicidade e a chamada da função *callback* para a chamada da função de execução ao recurso da Ikuzo API enquanto a Figura 4.3 demonstra a implementação dessa função que realiza uma requisição REST para o recurso de atualização de GPS da Ikuzo API.

O programa foi instalado na lista de serviços do sistema operacional do servidor Windows da aplicação principal para que sua execução seja mantida enquanto o servidor esteja em atividade.

[1] TimerCallback: https://docs.microsoft.com/pt-br/dotnet/api/system.threading.timercallback?view=netframework-4.8

```
using System.ServiceProcess;
using System.Threading;

namespace Ikuzo_bot
{
    public partial class IkuzoBot : ServiceBase
    {
        Timer _timer;

        public IkuzoBot()
        {
            InitializeComponent();
        }

        protected override void OnStart(string[] args)
        {
            _timer = new Timer(timer_Tick, null, 15000, 240000);
        }

        protected override void OnStop()
        {
        }

        private void timer_Tick(object sender)
        {
            var repo = new IntegrationRepository();
            repo.SyncGps();
        }
    }
}
```

Figura 4.2: Programa em C# para chamada por minuto do recurso da Ikuzo API de atualização de GPS

```
using System;
using System.Collections.Generic;
using System.Linq;
using RestSharp;

namespace Ikuzo_bot
{
    public class IntegrationRepository
    {
        protected readonly IRestClient Client;

        public IntegrationRepository()
        {
            var url = "http://ikuzo.eastus.cloudapp.azure.com/api/v1/";

            Client = new RestClient(url);
        }

        public IRestResponse SyncGps()
        {
            var request = new RestRequest("Crawler/Gps", Method.POST);

            Client.AddDefaultHeader("AccessKey", "51e17013-de05-4a7b-b917-8c35167c96e0");

            var response = Client.Execute(request);

            return response;
        }
    }
}
```

Figura 4.3: Classe C# para requisição REST do recurso da Ikuzo API de atualização de GPS

4.3 Mapeamentos

Após a integração aos repositórios de dados externos de transporte público, foi necessário estabelecer uma padronização dos dados para a carga no banco de dados da Ikuzo API. Nesta seção será descrito o mapeamento que foi adotado, campo a campo, entre a Ikuzo API e cada um dos repositórios.

4.3.1 Mapeamento de campos: Rio.bus

O repositório da Rio.bus disponibiliza, em formato JSON, dados com informações de linhas de ônibus e os modais que pertencem a linha, além de informação em tempo real de GPS.

O mapeamento realizado pode ser visto na tabela 4.1.

Classe	**Atributo na Ikuzo API**	**Campo no repositório Rio.bus**
Line	LineId	line
Line	Description	description
Modal	ModalId	order
Gps	Latitude	latitude
Gps	Longitude	longitude
Gps	Direction	direction
Gps	Timestamp	timestamp

Tabela 4.1: Mapeamentos realizados sobre dados do repositório Rio.bus

4.3.2 Mapeamento de campos: Data.rio

O repositório da Data.rio disponibiliza, em formato CSV, dados com informações itinerários e informação em tempo real de GPS.

O mapeamento realizado pode ser visto na tabela 4.2.

Classe	**Atributo na Ikuzo API**	**Campo no repositório Data.rio**
Gps	Latitude	LATITUDE
Gps	Longitude	LONGITUDE
Gps	Direction	DIRECAO
Gps	Timestamp	DATAHORA
Itinerary	Latitude	LATITUDE
Itinerary	Longitude	LONGITUDE
Itinerary	Sequence	SEQUENCIA

Tabela 4.2: Mapeamentos realizados sobre dados do repositório Data.rio

4.3.3 Campos calculados

Alguns campos foram calculados ou gerados pela própria Ikuzo API para compor a informação, sendo eles: LastUpdateDate nas classes Line, Modal, Itinerary e Gps, que contém a última data/hora que a Ikuzo API atualizou a informação; Returning, de Itinerary, qual nos baseamos na informação do campo Sequence para determinar se o ponto de itinerário faz parte do trajeto de ida ou de volta de uma linha; DistanceToNext, que é a distância em metros calculada através de Haversine (3.1) a fim de que seja possível estimar a distância total entre pontos de itinerários.

4.4 A aplicação Ônibus Carioca

A aplicação Ônibus Carioca foi desenvolvida sobre a plataforma Android com o intuito de ser uma simples interface gráfica dos dados provenientes da Ikuzo API integrada com os repositórios de dados de ônibus do Rio de Janeiro.

A Figura 4.4 a seguir mostra a arquitetura completa montada para o estudo de caso, onde a aplicação Android Ônibus Carioca realiza requisições para os recursos da Ikuzo API, o serviço do Windows responsável pelas atualizações dos dados da Ikuzo API além das integrações externas com a Rio.bus e a Data.rio que provisionam os dados sobre transporte público na cidade do Rio de Janeiro.

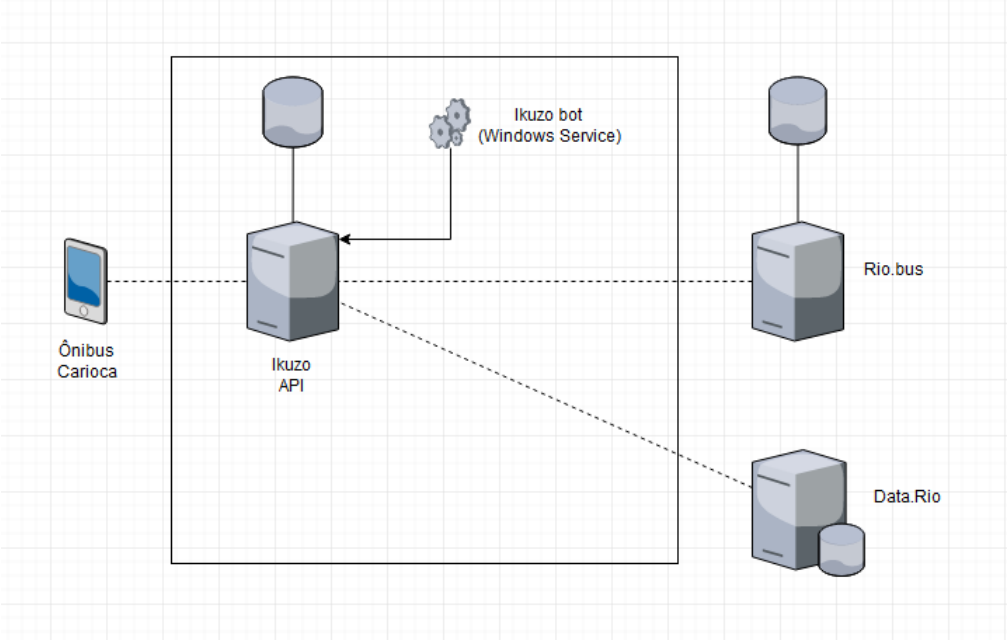

Figura 4.4: Arquitetura entre os sistemas integrados

4.4.1 Requisições à Ikuzo API

Para realizar as requisições REST para obtenção dos recursos disponibilizados pela Ikuzo API, foi utilizada a biblioteca OkHttpClient [2] em união com a biblioteca Gson [3] para a deserialização do JSON de retorno às requisições em classes Java da aplicação Ônibus Carioca.

4.4.2 Shared Preferences

Shared Preferences é uma biblioteca Android para persistência de pequenos dados, no formato de chave e valor, amplamente utilizado para guardar preferências do usuário sobre a aplicação. A aplicação Ônibus Carioca faz o uso dos recursos da biblioteca para guardar a preferência do usuário sobre a distância máxima que gostaria de obter os dados de GPS retornados da Ikuzo API.

4.4.3 Permissionamento

Alguns recursos ou bibliotecas Android somente podem ser utilizadas com o aceite do usuário. Para perfeita execução da aplicação Ônibus Carioca, é necessário que o usuário permita que o sistema acesse as informações do GPS do dispositivo para obtenção da posição atual do dispositivo e a permissão de acesso à internet para que possa ser estabelecida uma conexão sobre o protocolo HTTP entre a aplicação Ônibus Carioca e a Ikuzo API para obtenção das informações.

4.4.4 Arquitetura de desenvolvimento

A aplicação Ônibus Carioca foi modelada seguindo conceitos de MVC, dividindo as classes em pacotes:

- O pacote controller tem a função de orquestrar as requisições provenientes das *Activities* [4], fazendo requisições as classes repositórias e mapeando as respostas em classes do modelo;
- O pacote model contém as classes de domínio que são a abstração do contexto da aplicação;

[2] OkHttp: https://square.github.io/okhttp/
[3] Gson: https://github.com/google/gson
[4] Activities: https://developer.android.com/guide/components/activities.html

- O pacote repository é o responsável por se comunicar com a Ikuzo API, consumindo os recursos por ela disponibilizados;
- O pacote helper possui classes com métodos estáticos de auxílio a toda a aplicação;

A Figura 4.5 mostra a modelagem das classes da aplicação Ônibus Carioca seguindo o modelo UML.

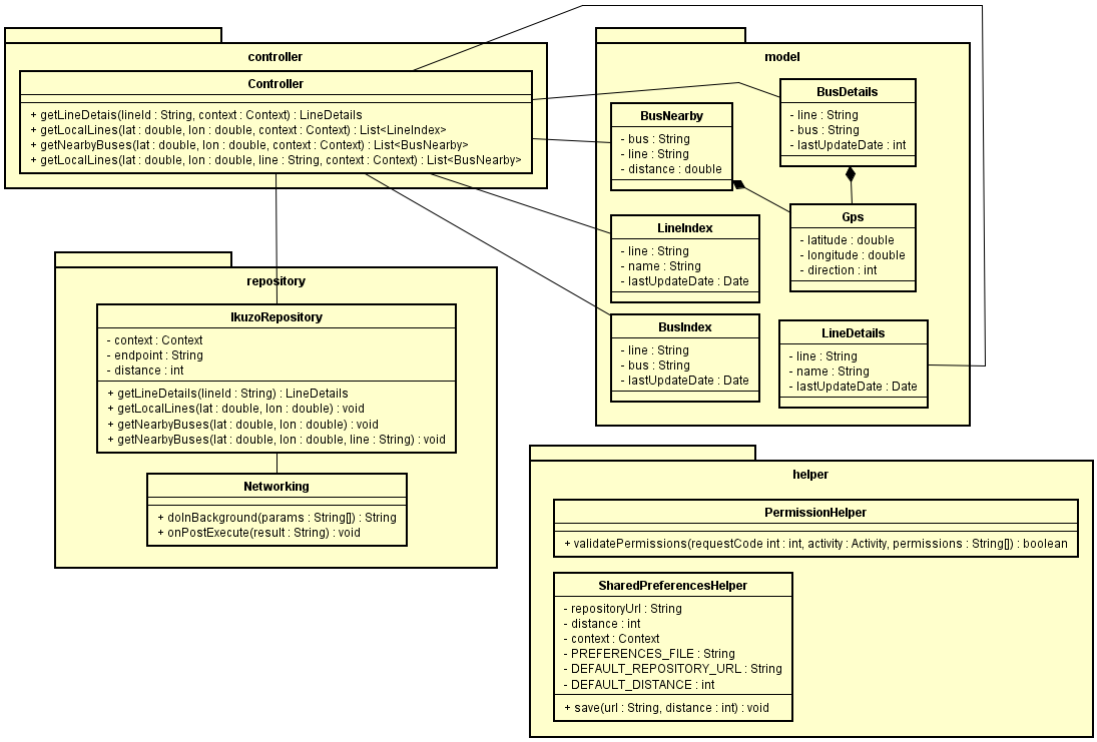

Figura 4.5: Diagrama de classes da aplicação Ônibus Carioca

4.4.5 Funcionalidades da aplicação Ônibus Carioca

A aplicação Ônibus Carioca foi integrada a Ikuzo API com o intuito de fornecer informações para o usuário de ônibus na cidade do Rio de Janeiro. As funcionalidades implementadas foram a consulta a ônibus próximos a um local, denominada Ônibus próximos, a consulta sobre modais de uma linha específica, informação também encontrada na área de Ônibus próximos e informações de itinerários de linhas de ônibus que passam por um determinado local. A aplicação ainda possibilita configurações sobre preferências de quão distante do local atual os resultados retornados devem ser.

A Figura 4.6 mostra a tela da aplicação contendo o menu com as funcionalidades disponíveis.

Figura 4.6: Menu de navegação da aplicação Ônibus Carioca

- Ônibus próximos

Realiza uma consulta na Ikuzo API baseado na latitude e longitude obtida do sistema de GPS do dispositivo Android do usuário. Será exibida uma lista de com informações de ônibus, ordenada pela proximidade do local do usuário. Também é possível realizar uma consulta por uma linha específica através de um campo de texto e um botão disponível na tela, caso feito, será retornada uma lista apenas com os ônibus da linha solicitada, também ordenada por proximidade do local obtido no dispositivo.

- Linhas deste local

Realiza uma consulta na Ikuzo API sobre quais linhas possuem itinerários próximos ao local obtido no GPS do dispositivo do usuário. Serão exibidas informações de todas as linhas que coincidirem com os requisitos. Ao clicar em uma das linhas, será redirecionado para a tela de Ônibus Próximos com a pesquisa dos ônibus da linha selecionada.

- Configurações avançadas

Permite ao usuário fazer configurações como alterar a URL para a conexão com a aplicação Ikuzo API bem como o alcance das buscas a serem realizadas. Ao clicar em salvar, essas informações de preferência serão salvas na memória do dispositivo através da biblioteca Shared Preferences. A Figura 4.7 mostra a página de configurações avançadas da aplicação Ônibus Carioca.

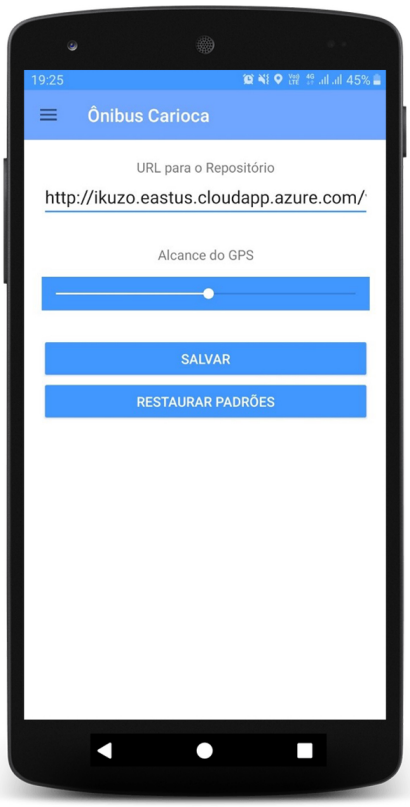

Figura 4.7: Configurações da aplicação Ônibus Carioca

4.5 Resultados obtidos e discussões

Nesta seção iremos analisar os resultados obtidos do estudo da aplicação Ikuzo API na cidade do Rio de Janeiro no contexto de transporte público no modal Ônibus. Os resultados serão aqui representados pelas telas da aplicação Android Ônibus Carioca, que serviu de interface para a Ikuzo API no estudo, e pela aplicação Postman [5] qual consegue fazer requisições via protocolo HTTP diretamente para a Ikuzo API, também mostraremos resultados quantitativos obtidos durante o período de testes.

4.5.1 Consultas simples de dados

Definimos como consultas simples de dados as consultas quais não são necessário informações georreferenciais nos parâmetros da requisição. São consultas que, basicamente, retornam as informações cadastradas no banco de dados da Ikuzo API.

[5] Postman Learning Center: https://learning.getpostman.com/

- Busca de linhas de ônibus

Consulta em todos os registros de Lines cadastrados no banco de dados. A Ikuzo API retornou com sucesso todas as linhas que foram extraídas dos repositórios o qual foi integrada para informações de linhas de ônibus da cidade Rio de Janeiro.

A Figura 4.8 mostra a tela do Postman qual realizou a requisição de linhas bem como a resposta em formato JSON da Ikuzo API.

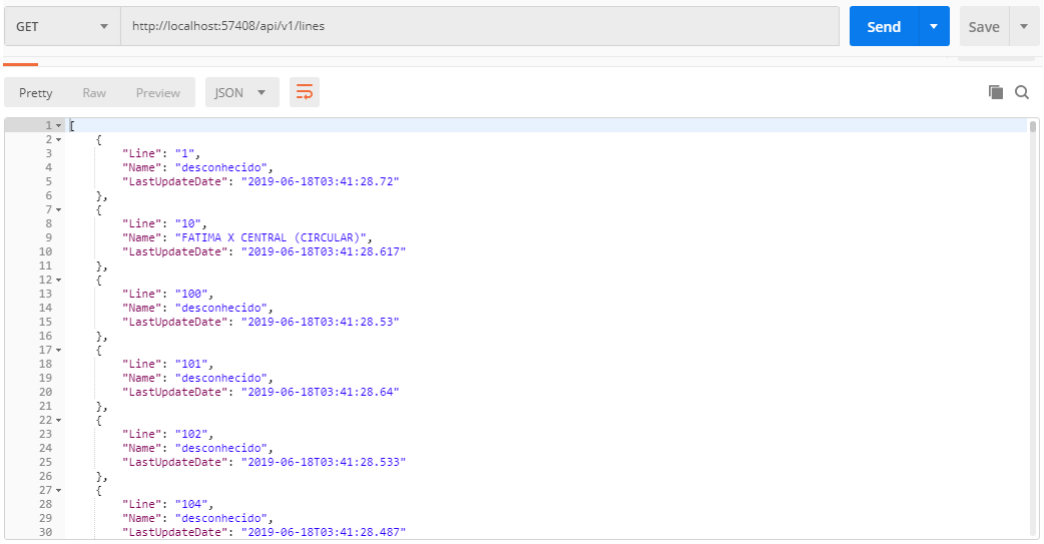

Figura 4.8: Busca de linhas na Ikuzo API

- Busca de linha de ônibus específica

Consulta orientada a um identificador de Line. A Ikuzo API retornou com sucesso os dados da linha 348 especificada bem como a informação de todos os registros de modais pertencentes a ela.

A Figura 4.8 mostra a mostra a tela do Postman qual realizou a requisição da linha 348 bem como a resposta em formato JSON da Ikuzo API

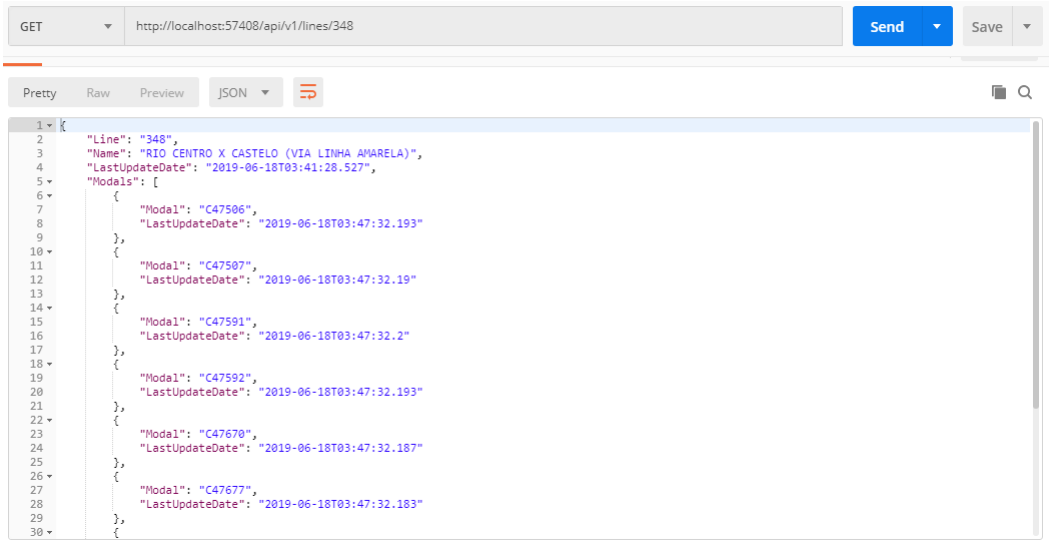

Figura 4.9: Busca de uma linha específica na Ikuzo API

- Busca de itinerário de uma linha de ônibus específica

Consulta orientada a um identificador de Line para busca de seu itinerário. A Ikuzo API retornou com sucesso as breves informações da linha de ônibus 348 bem como todo o seu itinerário com os pontos de latitude e longitude referentes.

A Figura 4.10 mostra a tela do Postman qual realizou a requisição de itinerários de uma linha bem como a resposta em formato JSON da Ikuzo API.

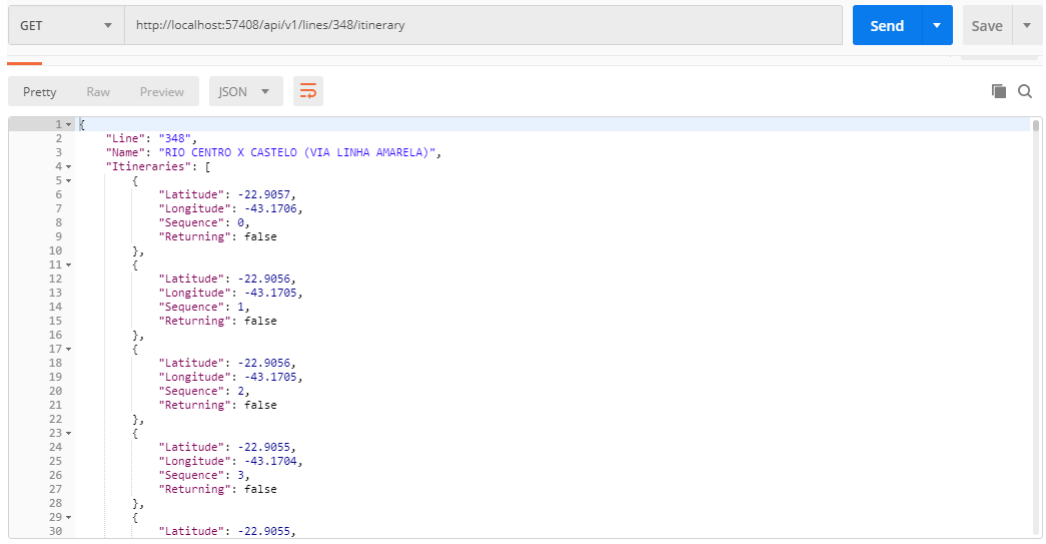

Figura 4.10: Busca de itinerário de uma linha na Ikuzo API

- Busca de ônibus

Consulta em todos os registros de Modals cadastrados no banco de dados. A Ikuzo API retornou com sucesso todos os registros que foram extraídos dos repositórios o qual foi integrada para informações de ônibus da cidade Rio de Janeiro.

A Figura 4.11 mostra a tela do Postman qual realizou a requisição de modais bem como a resposta em formato JSON da Ikuzo API.

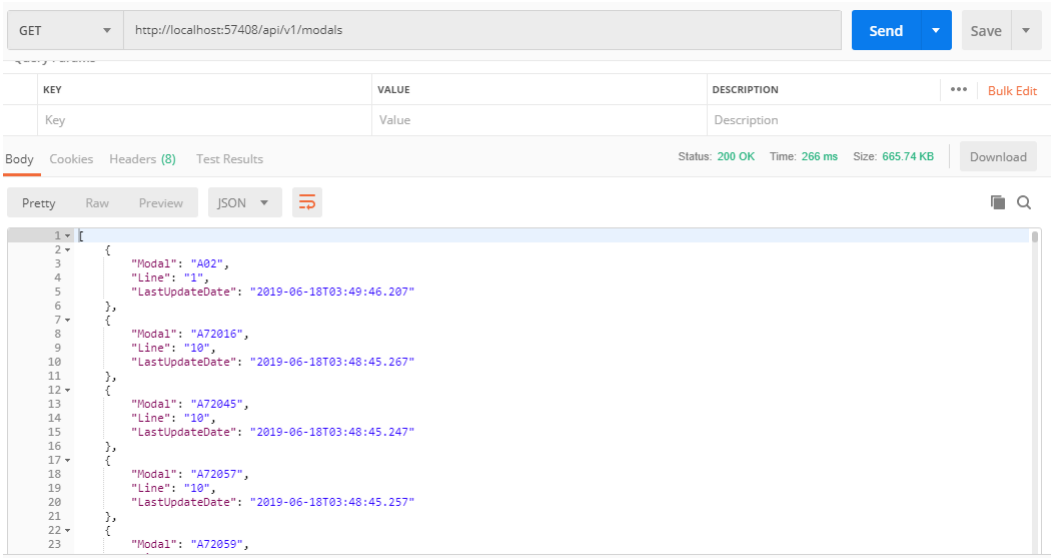

Figura 4.11: Busca de ônibus na Ikuzo API

- Busca de um ônibus específico

Consulta orientada a um identificador de modal. A Ikuzo API retornou com sucesso os dados do modal identificado por C47506, bem como a informação de Gps atual do mesmo no instante buscado.

A Figura 4.12 mostra a mostra a tela do Postman qual realizou a requisição do modal C47506 bem como a resposta em formato JSON da Ikuzo API.

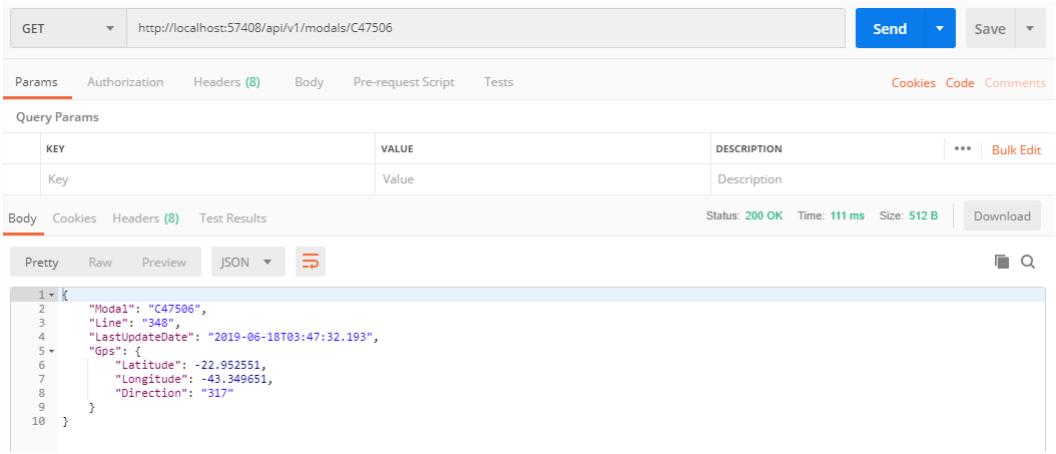

Figura 4.12: Busca de um ônibus específico na Ikuzo API

4.5.2 Consultas georreferenciadas

Definimos como consultas georreferenciadas as consultas quais é necessária a passagem de informações de latitude e longitude para a requisição e que retornam dados que se baseiam nos parâmetros e retornam dados calculados pela Ikuzo API.

- Ônibus próximos a um local

Consulta para busca de modais próximas a um ponto, realizada através da aplicação Ônibus Carioca, a Ikuzo API retornou com sucesso os ônibus que estavam próximos ao local Avenida Marechal Câmara, 314, no centro da cidade do Rio de Janeiro, latitude -22.9081057, longitude -43.1717964, quando foi feita a requisição.

A Figura 4.13 mostra a tela da aplicação Ônibus Carioca mostra o resultado da consulta realizada.

Figura 4.13: Tela com ônibus próximos ao ponto

- Linhas de ônibus com itinerário em um local

Consulta para busca de linhas que possuem itinerário próximas a um ponto, realizada através da aplicação Ônibus Carioca, a Ikuzo API retornou com sucesso as linhas que possuem itinerário próximo ao local Avenida Marechal Câmara, 314, no centro da cidade do Rio de Janeiro, latitude -22.9081057, longitude -43.1717964.

A Figura 4.14 mostra a tela da aplicação Ônibus Carioca mostra o resultado da consulta realizada.

Figura 4.14: Tela de linhas que possuem itinerário próximo ao ponto

- Planejamento de rota

Consulta para busca de linhas que possuem itinerário tanto ponto de origem quanto no ponto de destino, em um alcance para o estudo de caso de 200 metros, valor pré-estabelecido no servidor da Ikuzo API. Realizamos a consulta marcando como ponto de origem o bairro Centro da cidade do Rio de Janeiro, na latitude -22.9081057 e longitude -43.1717964 e como ponto de destino, o bairro de Jardim América, latitude -22.808066 e longitude -43.3237877 e a Ikuzo API nos retornou com sucesso duas linhas de ônibus que possuem itinerário nos dois pontos simultaneamente.

A Figura 4.15 mostra a consulta realizada pelo Postman bem como o resultado em formato JSON da Ikuzo API.

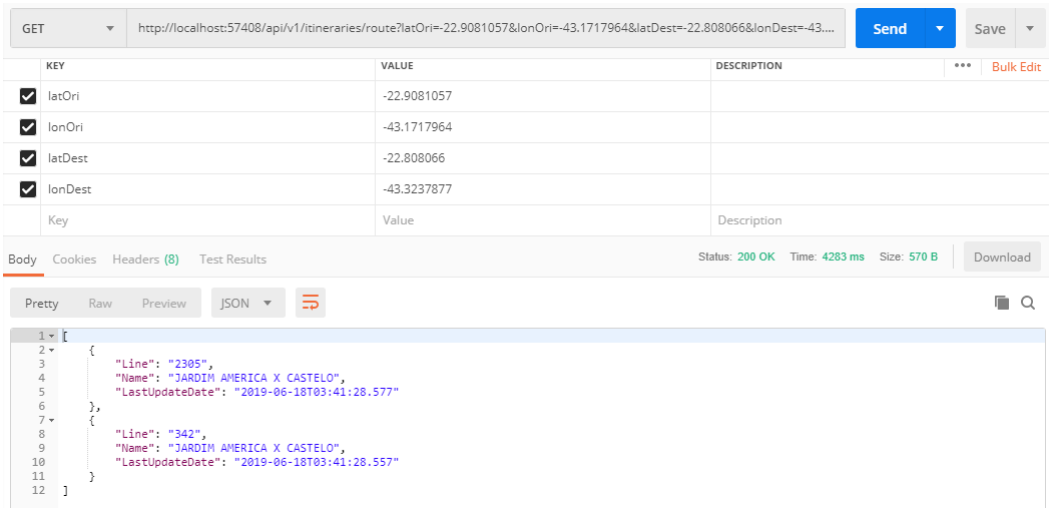

Figura 4.15: Busca de planejamento de rota na Ikuzo API

4.5.3 Resultados quantitativos

Aqui mostraremos resultados quantitativos obtidos durante execução do estudo de caso.

- Tempo de atualização de dados

A Tabela 4.3 mostra o tempo médio para a execução completa de cada solicitação de atualização de dados na Ikuzo API integrada nos repositórios externos Data.rio e Rio.bus. Cada recurso foi executado 20 vezes, de maneira sequencial e individual, sem concorrência entre as execuções, e no mesmo período do dia. O tempo apresentado se refere a todo o processo de extração dos dados dos repositórios, transformação e carga no banco de dados da Ikuzo API.

Recurso	**Quantidade de registros atualizados**	**Tempo médio de execução**
GPS	6570	2653 ms
Linhas	490	2760 ms
Modais	6569	167811 ms
Pontos de Itinerários	626481	159003 ms

Tabela 4.3: Tempo médio para atualização dos recursos

Com as informações da Tabela 4.3, pudemos estipular qual seria um tempo razoável para fazer as atualizações de informações na Ikuzo API: Uma vez que a atualização de todos os recursos de GPS, levou-se em torno de 2.6 segundos, devemos aplicar um tempo superior para realizar a próxima atualização, bem como para itinerários, onde o tempo foi

de aproximadamente 159 segundos. Neste segundo também devemos levar em consideração a semântica do recurso: Um itinerário não deve ser atualizado a todo momento, pois não é algo sensível a tempo real, e, portanto, não deverá sofrer atualizações frequentes.

O tempo para atualização dos recursos são bastante influenciáveis pelo tempo de resposta dos repositórios de origem dos dados Data.rio e Rio.bus.

4.5.4 Estudos em dados históricos

Para nosso estudo de caso, ao ser integrada com os repositórios de dados de informação de geolocalidade de ônibus, foi programado a atualização dos dados na Ikuzo API a cada 3 minutos. Dada periodicidade fez com que a tabela GpsHistory do banco de dados fosse sendo acumulados para que pudessem ser realizados análises sobre eles para verificarmos possíveis defasagem de dados e propor novos trabalhos para aumentar a capacidade da Ikuzo API. Mais informações relacionadas a trabalhos futuros serão vistas no próximo capítulo.

Aqui faremos a análise de resultados para o comportamento de um modal selecionado para um dia em específico, durante vários horários deste dia, verificaremos sua corretude com o trajeto determinado pelas informações de itinerário, analisaremos tempos de trajeto e, por fim, compararemos duas linhas com trajetos semelhantes.

- Análise nos dados georreferenciais de um modal

Selecionamos da base de dados o modal de identificação A63536 pertencente a linha 107 para realizarmos este estudo. Analisamos as informações históricas de GPS do dia 10 de dezembro de 2019, durante o período do dia de 11:45 a 22:50. A linha faz o trajeto Central x Urca.

Dado que fizemos o processo de extração de dados em dois repositórios diferentes, é importante validarmos sua coesão. A Figura 4.16 mostra todos os pontos de GPS que obtivemos durante nossas atualizações da posição do modal, e ao compararmos com o trajeto definido pelo itinerário, mostrado na Figura 4.17, percebemos que, de fato, o modal seguiu o trajeto cadastrado em nosso banco de dados. Os pontos adicionais fora do trajeto padrão, que podem ser identificados pela menor quantidade de pontos marcados no mapa, acontecem pelo trajeto que o modal percorreu até sua garagem.

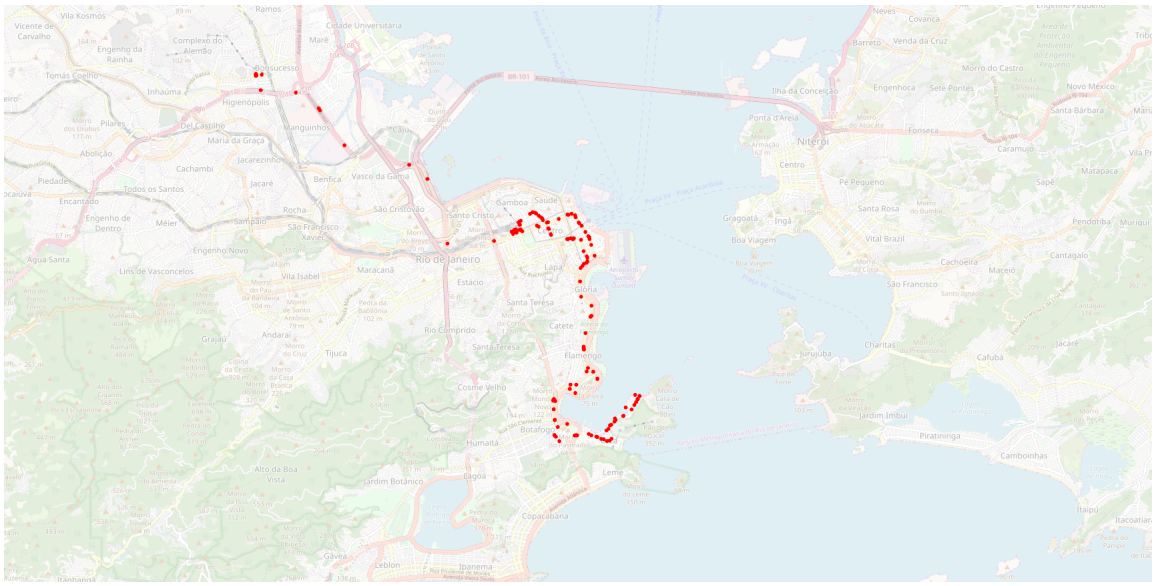

Figura 4.16: Pontos de GPS coletado da linha 107 no período analisado

Figura 4.17: Itinerário completo cadastrado da linha 107

Viagem	**Direção**	**Início**	**Fim**	**Tempo percorrido**	**Tempo total da viagem**
Viagem 1	Ida	12:43	14:06	01:23:00	01:44:00
	Volta	14:24	14:45	00:21:00	
Viagem 2	Ida	14:57	16:09	01:12:00	02:00:00
	Volta	16:09	16:57	00:48:00	
Viagem 3	Ida	17:00	18:13	01:13:00	01:52:00
	Volta	18:15	18:54	00:39:00	

Tabela 4.4: Horários e tempos de viagem do modal A63536 da linha 107

Ainda com o modal A63536, conseguimos levantar alguns tempos de viagem que levou durante o dia afim de identificarmos horários de pico de trânsito, entre outras informações. A Tabela 4.4 mostra as análises dos tempos de viagem, onde uma viagem é definida como todo o trajeto que o modal percorre, ida e volta, do bairro do Centro até o bairro da Urca. Para se identificar o momento em que uma viagem se inicia ou termina, usamos a fórmula de Haversine para verificar o ponto que mais se aproximava do primeiro ou do último ponto da sequência do itinerário da linha.

Com essas informações podemos reparar alguns padrões da linha 107, como que o tempo total de viagem não teve uma mudança muito perceptível em algum horário específico do dia, dado horário de maior movimento no trânsito. A maior diferença foi de 16 minutos entre a primeira e a segunda viagem.

O fato que nos chamou atenção foi o retorno da viagem ao Centro ser bem mais rápido que a ida. O comportamento é explicado pelo fato de que a linha segue por vias expressas em seu retorno ao bairro Centro. Portanto, esses estudos servem também para identificar e sugerir linhas que tenham seus trajetos mais rápidos que outras na funcionalidade de planejamento de rotas da Ikuzo API. Mais detalhes serão vistos no capítulo seguinte, na área de trabalhos futuros. No exemplo estudado, a linha 107 pode ser mais sugerida no retorno ao Centro do que para a ida a Urca se o critério for menor tempo de viagem.

Agora selecionamos modais de outras duas linhas com a finalidade de compará-los, dado que possuem as mesmas origem e destino, verificarmos se é possível inferir horários que uma linha chega ao seu destino mais rapidamente que a outra. Essa análise foi feita nos modais C27120 e C27046, respectivamente das linhas 774 e 775, ambas com o itinerário Jardim América X Madureira. As Figuras 4.18 e 4.19 mostram o itinerário cadastrado das linhas 774 e 775, na ordem.

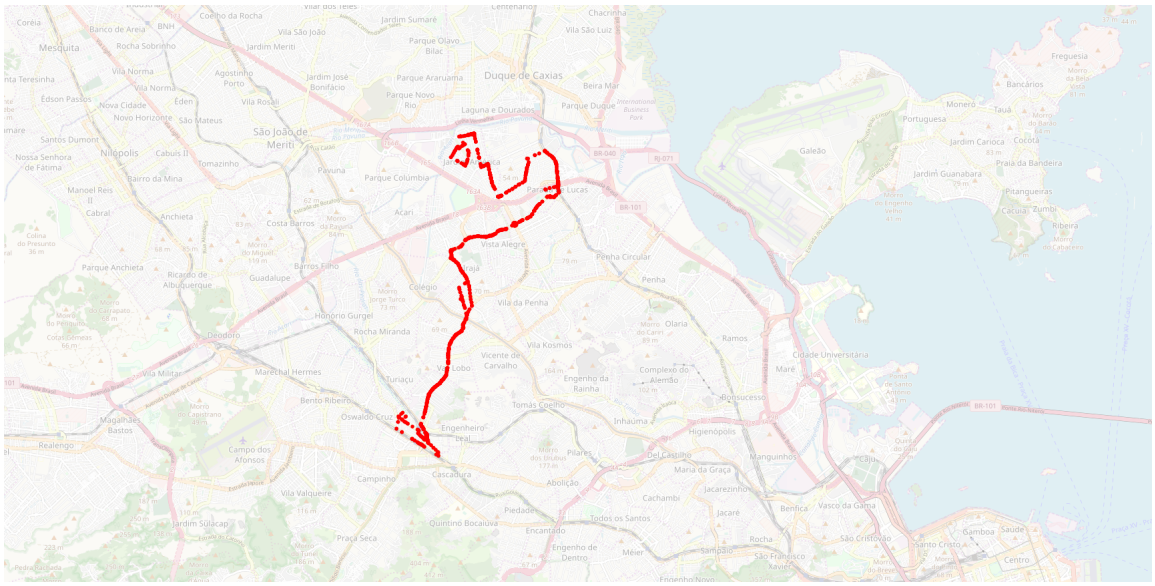

Figura 4.18: Itinerário completo cadastrado da linha 774

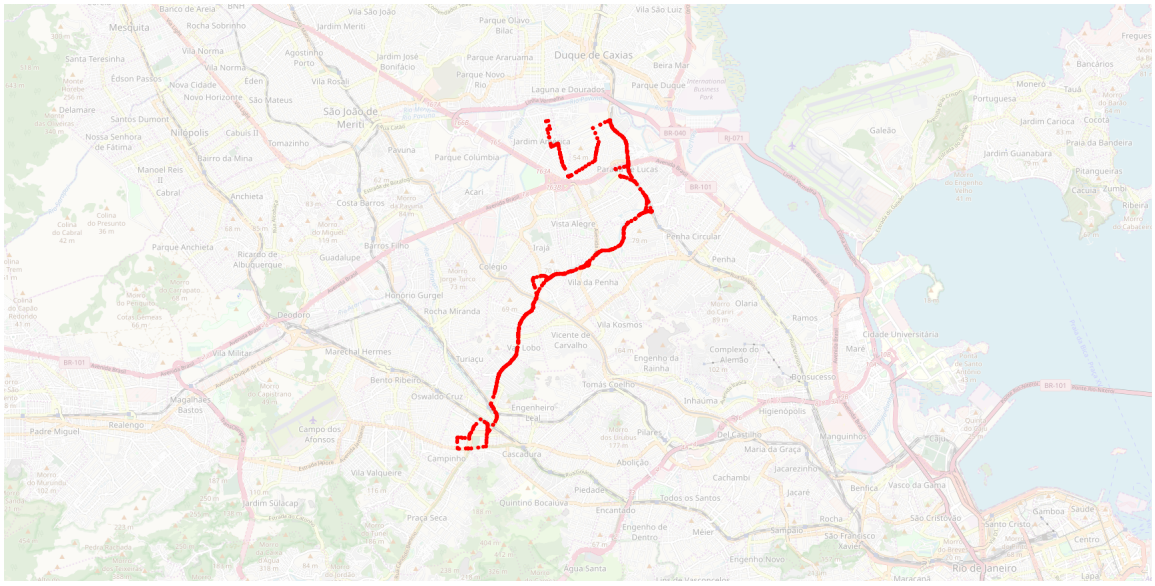

Figura 4.19: Itinerário completo cadastrado da linha 775

Viagem	Direção	Início	Fim	Tempo percorrido	Tempo total da viagem
Viagem 1	Ida	13:21	14:57	01:36	02:51:00
	Volta	15:00	16:15	01:15	
Viagem 2	Ida	16:30	17:51	01:21	02:48:00
	Volta	17:54	19:21	01:27	
Viagem 3	Ida	19:36	20:42	01:06	02:12:00
	Volta	20:45	21:51	01:06	

Tabela 4.5: Horários e tempos de viagem do modal C27120 da linha 774

Viagem	Direção	Início	Fim	Tempo percorrido	Tempo total da viagem
Viagem 1	Ida	16:15	17:30	01:15	02:39:00
	Volta	17:48	19:12	01:24	
Viagem 2	Ida	19:15	20:12	00:57	02:39:00
	Volta	20:39	22:21	01:42	
Viagem 3	Ida	22:24	23:18	00:54	02:42:00
	Volta	23:36	01:24	01:48	

Tabela 4.6: Horários e tempos de viagem do modal C27046 da linha 775

Levantamos os tempos de viagem dos modais das linhas citadas para comparação. O tempo total de viagem de ambos os modais, apesar de serem linhas que tomam caminhos diferentes, mas que possuem mesma origem e destino não foram muito distintos. Os dois modais se encontravam próximos ao seu ponto de início de trajeto, no bairro do Jardim América, no momento do primeiro percurso. As Tabelas 4.5 e 4.6 mostram os resultados da análise para os modais C27120 e C27046 respectivamente.

O ponto que conseguimos notar é que, apesar do modal da linha 775 se manter com um tempo médio de percurso menor que a linha 774 durante a maior parte do tempo analisado, durante o seu último percurso teve um tempo 30 minutos acima da linha 774. A coleta e estudo destes dados se tornam importante para o momento em que se pensa em uma implementação de aprendizado de máquina sobre eles para que sejam possíveis predições como sobre qual linha normalmente chegaria mais rápido naquele período do dia, naquele dia da semana. Para uma implementação do gênero seria necessário o estudo de uma grande quantidade de dados em um espaço temporal razoável, diferentemente do estudo aqui feito. Falaremos mais sobre essas implementações na área de trabalhos futuros, no próximo capítulo.

4.6 Considerações finais

Nesta seção foi demonstrada a utilização da Ikuzo API na cidade do Rio De Janeiro, realizando integrações com repositórios externos de dados abertos de transporte urbano

da cidade. Vimos a implementação da aplicação Android Ônibus Carioca como interface para os dados da Ikuzo API. Todo o código fonte da aplicação Ônibus Carioca é aberto e pode ser encontrado em https://github.com/bruno-sales/Onibus-Carioca. Também tivemos resultados satisfatórios quanto ao estudo da Ikuzo API aplicada na cidade do Rio de Janeiro, sendo obtidos os resultados esperados, além das informações quantitativas da execução e a qualidade nos dados que a aplicação nos proporciona para estudos sobre o transporte, a auxiliar na mobilidade urbana.

No capítulo a seguir faremos algumas observações bem como a conclusão do estudo aqui proposto.

5. Conclusão

Falaremos sobre as conclusões do trabalho apresentado neste capítulo, fazendo uma breve retrospectiva sobre seu objetivo e como foi aplicado o estudo de caso. Também falaremos por fim sobre trabalhos futuros originados deste trabalho.

5.1 Considerações gerais

O trabalho apresentado tinha por objetivo o desenvolvimento de uma API para apoio a mobilidade urbana no contexto de transportes públicos. Com isso, desenvolvemos a Ikuzo API onde, através de técnicas de ETL, é disponibilizado um repositório sobre informações de geolocalização em quase tempo real, dependendo da periodicidade configurável para atualização dos dados.

Para estudar o comportamento da aplicação, a aplicamos na cidade do Rio de Janeiro, no meio de transporte público ônibus. Integramos a aplicação em dois sistemas abertos para extração dos dados: O Data.Rio, repositório da prefeitura da cidade, e o Rio.bus. Com isso, os dados obtidos foram transformados e cadastrados em um banco de dados local da aplicação Ikuzo API e para interface gráfica destes dados, criamos ainda uma aplicação Android chamada Ônibus Carioca que faz requisições REST a Ikuzo API para a disponibilização dos dados, fazendo assim uma comunicação cliente-servidor.

Tivemos resultados satisfatórios quanto à qualidade e consistência dos dados na Ikuzo API, ponto importante dado que a escolha dos repositórios de dados para a extração, aqui utilizados o Data.rio e o Rio.bus, influenciam em toda a resolução de nossa aplicação. Outro ponto importante se mostrou a importância da resiliência na aplicação: Durante os estudos, por vezes os repositórios Data.Rio e Rio.bus se mostraram indisponíveis para obtenção dos dados, neste cenário a Ikuzo API teve de ignorar a solicitação de atualização dos dados, mantendo seus dados até que fosse possível uma extração de informação

dos repositórios externos a fim de não comprometer aplicações cliente, como a Ônibus Carioca.

5.2 Trabalhos futuros

Identificamos oportunidades para trabalhos futuros no estudo que foi aplicado, dentre elas está o uso de distância urbana - distância entre ruas, seguindo por estradas - ao invés do uso da fórmula de Haversine que, apesar de nos dar a distância entre dois pontos de latitude e longitude, não leva em consideração ruas paralelas, por exemplo. A distância entre dois pontos que se encontram em duas ruas paralelas maior caso utilizado a distância urbana entre os locais. A verificação das condições de trânsito para a sugestão também é um trabalho de aprimoramento válido.

A adição de novos repositórios de dados sobre transporte público também é um possível novo trabalho, seja para implementar novas contingências caso algum repositório esteja indisponível, para validação dos dados, ou mesmo para obtenção de novos dados, como para as informações de Tags, classificando as linhas de acordo com alguma base de dados pública.

Outro trabalho futuro que conseguimos levantar é, dado a quantidade de dados que passa pela Ikuzo API, a possibilidade de fazer estudos de *Machine Learning* para predições e assim melhorar funcionalidades da aplicação como o planejamento de rotas: Caso uma linha, historicamente em um dado horário do dia, leva mais tempo que uma outra linha, essa segunda deve ser apresentada com mais relevância que a primeira. Para a implementação dessa funcionalidade, é sugerido também que seja alterado o banco de dados da aplicação. O SQL Server se mostrou eficaz para o armazenamento dos dados, porém, sem a criação de muitos índices em suas tabelas, consultas com um grande período temporal como as realizadas em uma funcionalidade de *Machine Learning* necessitaria, tornaria um grande desafio de ser realizada em tempo hábil. Para tal funcionalidade seria recomendado o uso de bancos de dados não relacionais ou um *data warehouse* específico para esse tipo de trabalho.

Referências Bibliográficas

[1] United Nations, Departament of Economic and Social Affairs. (2018). World Urbanization Prospects: The 2018 Revision.

[2] Instituto Brasileiro de Geografia e Estatística. (2015). Pesquisa Nacional por Amostra de Domicílios (PNAD).

[3] Rosana Melo de Araújo, Marley. (2011) Transporte público coletivo: discutindo acessibilidade, mobilidade e qualidade de vida. ISSN 1807-0310.

[4] Rahm, Erhard & Hai Do, Hong. (2000). Data Cleaning: Problems and Current Approaches. IEEE Data Eng. Bull.. 23. 3-13.

[5] Khan, Sakib M. & Rahman, Mizanur & Apon, Amy & Chowdhury, Mashrur. (2017). Characteristics of Intelligent Transportation Systems and its Relationship with Data Analytics.

[6] Trujillo, Juan & Luján-Mora, Sergio. (2003). A UML Based Approach for Modeling ETL Processes in Data Warehouses. 2813. 307-320. 10.1007/978-3-540-39648-2_25.

[7] Sohan, S.M. & Anslow, Craig & Maurer, Frank. (2015). A Case Study of Web API Evolution. 245-252. 10.1109/SERVICES.2015.43.

[8] Fielding, Roy Thomas. (2000) Architectural Styles and the Design of Network-based Software Architectures. Doctoral dissertation, University of California, Irvine.

[9] Vaqqas, M. (2014). RESTful Web Services: A Tutorial. Disponível em: <http://www.drdobbs.com/web-development/restful-web-services-a-tutorial/240169069>

[10] Sundvall, Erik & Nyström, Mikael & Karlsson, Daniel & Eneling, Martin & Chen, Rong & Orman, Håkan. (2013). Applying representational state transfer (REST) architecture to archetype-based electronic health record systems. BMC medical informatics and decision making. 13. 57. 10.1186/1472-6947-13-57.

[11] Jones, Matthew (2015). Dapper vs Entity Framework vs ADO.NET Performance Benchmarking. Disponível em <https://exceptionnotfound.net/dapper-vs-entity-framework-vs-ado-net-performance-benchmarking>

[12] Leach, Paul J., Salz Rich & Mealling, Michael H. (2005). RCF 4122: A Universally Unique IDentifier (UUID) URN Namespace. Disponível em <https://tools.ietf.org/html/rfc4122>

[13] Fielding, Roy Thomas & Reschke, Julian (2014). RCF 7235: Hypertext Transfer Protocol (HTTP/1.1): Authentication. Disponível em <https://tools.ietf.org/html/rfc7235>

[14] Saint-Andre, Peter & Klensin, John C. (2017). RCF 8141: Uniform Resource Names (URNs). Disponível em <https://tools.ietf.org/html/rfc8141>

[15] Jackson, Jim (2009). Calculating Distances Between Latitude and Longitude – T-SQL – Haversine. Disponível em <https://weblogs.asp.net/jimjackson/calculating-distances-between-latitude-and-longitude-t-sql-haversine>

[16] Bertoche, Uriel (2016). Sistema de Apoio à Mobilidade Urbana com Dados Abertos. Disponível em <https://bsi.uniriotec.br/wp-content/uploads/sites/31/2020/05/201612UrielBertoche.pdf>

[17] Data.rio - Dados abertos - Prefeitura da cidade do Rio de Janeiro. Disponível em <http://dadosabertos.rio.rj.gov.br/>

[18] Dados de transporte público da cidade do Rio de Janeiro - Rio.bus. Disponível em <http://riob.us/>