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

ELISA DE FÁTIMA ANDRADE SOARES

API URB: UMA API REST PARA GERENCIAMENTO DE PROBLEMAS URBANOS

MOSSORÓ - RN

2019

ELISA DE FÁTIMA ANDRADE SOARES

API URB: UMA API REST PARA GERENCIAMENTO DE PROBLEMAS URBANOS

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

MOSSORÓ - RN

2019

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

A553a Andrade Soares, Elisa de Fátima
API URB: UMA API REST PARA GERENCIAMENTO DE PROBLEMAS URBANOS. / Elisa de Fátima Andrade Soares. - Mossoró, 2019.
86p.

Orientador(a): Prof. Dr. Sebastião Emídio Alves Filho.
Monografia (Graduação em Ciência de Computação). Universidade do Estado do Rio Grande do Norte.

1. Problemas Urbanos, REST, Web Services, Taxonomia URB.. I. Alves Filho, Sebastião Emídio. II. Universidade do Estado do Rio Grande do Norte. III. Título.

O serviço de Geração Automática de Ficha Catalográfica para Trabalhos de Conclusão de Curso (TCC´s) foi desenvolvido pela Diretoria de Informatização (DINF), sob orientação dos bibliotecários do SIB-UERN, para ser adaptado às necessidades da comunidade acadêmica UERN.

ELISA DE FÁTIMA ANDRADE SOARES

API URB: Uma API REST para gerenciamento de problemas urbanos

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

Aprovada em: 12/07/2019

Banca Examinadora

Prof. Dr. SEBASTIÃO EMÍDIO ALVES FILHO
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. Dr. Rommel Wladimir de Lima
Universidade do Estado do Rio Grande do Norte - UERN

Aos meus pais, em especial a minha mãe.

Agradecimentos

Agradeço à Deus em primeiro lugar, A Ele a Glória, a Ele o Louvor, Ele é o meu Senhor.

À Maria Santíssima, a qual tenho uma grande devoção e sou consagrada.

Quero agradecer a minha família, em especial a minha mãe, Maria de Fátima Andrade Soares que esteve comigo em todos os momentos difíceis sempre orando por mim, ao meu pai Etvaldo Soares Irmão, à minha tia Maria da Salete Soares e aos meus irmãos Elton Andrade Soares e Elvis Andrade Soares.

As minhas amigas que sempre acreditaram em mim, mas quero dá ênfase a Shayelli Laiany Mareco Abrantes que mesmo distante a todo momento esteve comigo me incentivando e apoiando.

Aos meus amigos que adquiri durante a graduação, em destaque a Leonardo Bandeira de Lucena e Sebastião Mateus Marques de Menezes, por todos os auxílios em diversos momentos. À Ligia Maria de Sousa Dantas Batista que esteve presente desde do início nesta etapa que me deu forças e cuidou de mim durante os dois anos de convivência e por toda a paciência. À Thiago de Oliveira Pereira que também cuidou de mim durante esses anos e sempre esteve ao meu lado me ajudando e dando conselhos. À Thalia Katiane Sampaio Gurgel que me deu todo apoio necessário ao chegar no curso e a todos os momentos que esteve presente. À Exlley Clemente dos Santos, por todos os auxílios e parceira nas monitorias.

À Claudivan Barreto da Silva, Adriano Ferreira Santos, João Paulo Silva de Moura, Wilton Silva dos Santos Júnior, Giovana Lorena Costa de Andrade e Álvaro Gabriel Gomes de Oliveira por te me ajudado durante a graduação.

À Universidade do Estado do Rio Grande do Norte. Ao Programa de Educação Tutorial de Ciência da Computação da UERN e todos os petianos que convivi e que me proporcionaram grandes ensinamentos e bons momentos.

Aos docentes do departamento de Informática, em especial Rommel Wladmir de Lima, o qual sempre me apoiou, incentivou e acreditou em mim, o qual tive a oportunidade de ser sua aluna e trabalhar no PETCC. Ao meu Orientador Sebastião Emidio Alves Filho por toda a sua dedicação e atenção na orientação deste trabalho. Ao professor Francisco Chagas de Lima Júnior, pela participação no projeto de Iniciação Científica no Laboratório de Otimização e Inteligência Artificial (LOIA). Ao Professor Marcelino Pereira dos Santos Silva, o qual era tutor do PETCC quando ingressei. À professora Alexsandra Ferreira Gomes que me deu a oportunidade de trabalhar no projeto de extensão Reaproveitamento

do Lixo Tecnológico e a por todos os conselhos para a vida. Ao professor Dario José Aloise, pela oportunidade de ser monitora da disciplina de Teoria dos Grafos, por sempre acreditar em mim. À Professora Ceres Germanna Braga Morais, a qual trabalhei juntamente na Comissão Setorial de Avaliação Institucional, e que sempre ajudou nas correções de artigos. Ao Professor André Pedro Fernandes Neto pela oportunidade de estágio, que foi em um curto período mas que fez parte da minha formação. À Professora Carla Katarina de Monteiro Marques, por sempre confiar no meu trabalho. Ao Professor Everton Notreve Rebouças Queiroz, que trabalhei na organização da I Semana Acadêmica de Ciência da Computação. À professora Cícilia Raquel Maia Leite, a qual trabalhei juntamente na Semana de Ciência da Computação e por todo apoio para antecipação deste trabalho. Ao professor Pedro Fernandes Ribeiro Neto, por ter aceitado o meu convite para ministrar a disciplina de Métodos Formais e por todo o conhecimento. À professora Jéssica Neiva de Figueiredo Leite por todo apoio e conhecimento.

Aos técnicos do departamento de Informática, em especial ao meu amigo Mizael Clistion Souza Elias e Elis Emídia Rebouças De Carvalho que me ajudaram bastante.

Ao Centro Acadêmico de Ciência da Computação.

Por fim, agradeço à todos a qual não citei, mas que contribuíram de forma direta ou indireta para a minha formação.

"Talvez não tenha conseguido fazer o melhor, mas lutei
para que o melhor fosse feito."
(Marthin Luther King)

Resumo

A expansão das cidades, o processo de urbanização, o crescimento populacional e a falta de planejamento são fatores que contribuem para o surgimento de problemas urbanos. Tendo em vista que o poder público é o maior responsável pela solução desses problemas, proporcionar uma infraestrutura adequada para participação do cidadão na gestão urbana é fundamental. Contudo, é comum encontrar uma variedade de meios de comunicação e sistemas de informação para esse fim. Em várias oportunidades esses sistemas não são integrados, nem possibilitam um acompanhamento da solução ao longo do tempo. Nesse intuito, emerge a necessidade de recursos que possam auxiliar nesse processo. Para isso, este trabalho propõe uma API REST, denominada de API URB, com o intuito de gerenciar as informações geradas a partir dos relatos de problemas urbanos pelos cidadãos. A API possibilita uniformizar o acesso aos dados armazenados e viabiliza o acompanhamento do histórico de relatos e resoluções dos mesmos. A API URB segue os princípios do estilo arquitetural REST e faz uso de *Web Services*, permitindo que aplicações implementadas em diferentes plataformas e linguagens de programação possam consumir os serviços. Além disso, este estudo apresenta um modelo a ser adotado em outras aplicações de relatos urbanos, baseado em uma taxonomia, denominada de Taxonomia URB, que consiste em apresentar os problemas urbanos divididos em categorias. A implementação da API faz uso da linguagem Javascript e da plataforma Node.js por meio do auxílio do *framework* Express e o ArangoDB como banco de dados, além de dispor de documentação com as rotas implementadas. A validação foi realizada através da aplicação Reporte Cidadão com sua plataforma web e *mobile*. O desenvolvimento deste trabalho possibilita a utilização de *Web Services* com um bom desempenho, agilidade e escalabilidade para poder fornecer os serviços implementados a uma grande quantidade de usuários, possibilitando uma melhor interação entre cidadãos e poder público.

Palavras-chave: Problemas Urbanos, REST, Web Services, Taxonomia URB.

Abstract

The expansion of cities, the process of urbanization, population growth and lack of planning are factors that contribute to the emergence of urban problems. In view of that public power is the most responsible for solving these problems, providing adequate infrastructure for citizen participation in urban management is crucial. However, it is common to find a variety of media and information systems for this purpose. On several occasions, these systems are not integrated, nor do they make it possible to monitor the solution over time. In this way, the need for resources that may help in this process emerges. For this, this work proposes a REST API, called URB API, in order to manage the information generated from reports of urban problems by citizens. The API makes it possible to standardize the access to the stored data and enables the monitoring of the reports historic and resolutions of the same. The URB API follows the principles of the REST architectural style and makes use of Web Services, allowing applications deployed across different platforms and programming languages to consume the services. In addition, this study presents a model to be adopted in other applications of urban reports, based on a taxonomy, called Taxonomy URB, which consists of presenting the urban problems divided into categories. The API implementation makes use of the Javascript language and the Node.js platform through the help of the Express framework and ArangoDB as a database, in addition to having documentation with the routes implemented. The validation was done through the Citizen Reporte application with its web and mobile platform. The development of this work allows the use of Web Services with good performance, agility and scalability to be able to provide the services implemented to a large number of users, enabling a better interaction between citizens and public power.

Keywords: Urban Problems, REST, Web Services, Taxonomy URB.

Lista de ilustrações

Lista de códigos

Lista de tabelas

Lista de abreviaturas e siglas

ANSI	American National Standards Institute
API	Application Programming Interface
AQL	Arango Query Language
CORBA	Common Object Request Broker Architecture
DCOM	Distributed Component Object Model
FIFO	First In, First Out
HATEOAS	Hypermedia As The Engine Of Application State
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
HTTPS	Hyper Text Transfer Protocol Secure
IBGE	Instituto Brasileiro de Geografia e Estatística
IEEE	Instituto de Engenheiros Elétricos e Eletrônicos
JSON	JavaScript Object Notation
JWT	Json Web Token
MVC	Model-view-controller
NOSQL	Not Only SQL
NPM	Node Package Manager
OSI	Open System Interconnection
REST	Representational State Transfer
RFC	Request for Comments
RMI	Remote Method Invocation
RPC	Remote Procedure Call
SAAS	Software as a service

SOA	Service Oriented Architecture
SOAP	Simple Object Access Protocol
TCP	Transmission Control Protocol
TI	Tecnologia da Informação
TIC	Tecnologia da Informação e Comunicação
UDDI	Universal Description, Discovery and Integration
URI	Uniform Resource Identifier
URL	Universal Resource Locator
W3C	World Wide Web Consortium
WSDL	Web Services Description Language
XML	eXtensible Markup Language

Sumário

1 Introdução

Dentre os processos pelos quais as cidades passam, existe o de urbanização. Esse termo geográfico é utilizado para descrever o desenvolvimento das cidades e o crescimento populacional, marcado pela transição da população da zona rural para a urbana (RIBEIRO; VARGAS, 2015). Segundo Santos (2005) a urbanização no Brasil iniciou-se no século XX, ocorrendo de maneira acelerada, diferentemente dos países de primeiro mundo, que passaram entre cem e duzentos anos na Revolução Industrial. No caso do Brasil, esse processo durou aproximadamente cinquenta anos, tendo como motivação a procura de emprego.

O processo de urbanização provoca mudanças no espaço urbano no que tange às questões sociais, econômicas, culturais e ambientais. Elas podem prejudicar a qualidade de vida dos habitantes em aspectos pessoais e coletivos, sendo muitos deles de responsabilidade do poder público. O crescimento populacional é outra consequência. Segundo o último censo demográfico, realizado em 2010, pelo Instituto Brasileiro de Geografia e Estatística (IBGE), 160.925.792 brasileiros (84,4% da população) já residiam em áreas urbanas, contra apenas 29.852.986 (15,6%) que moravam em áreas rurais.

As cidades crescem de maneira rápida e junto a este crescimento surge a necessidade de realizar um planejamento urbano, que é um processo essencial à obtenção da qualidade de vida da população e na formação de uma área urbana. Esse processo abrange a estruturação de um espaço urbano, com a finalidade de auxiliar e melhorar a gestão urbana. Segundo Souza e Rodrigues (2004) o planejamento urbano e a gestão urbana são atividades distintas. O planejamento urbano refere-se às estratégias elaboradas para serem executadas para o futuro, uma forma de prevenir possíveis problemas, enquanto a gestão urbana consiste em técnicas utilizadas para administrar as cidades.

Conforme Clezar (2006) o desenvolvimento das cidades está correlacionado aos serviços de infraestrutura. A falta de infraestrutura, por exemplo, dificulta a locomoção com segurança e conforto dos indivíduos. Almeida, Giacomini e Bortoluzzi (2013) concluem que não têm surgido projetos de mobilidade urbana voltados para as pessoas e, sim, para automóveis. Ao invés de melhorar a acessibilidade, com calçadas mais largas e um maior número de ciclovias, ou ainda buscar uma cidade mais sustentável com parques e áreas verdes, estão sendo construídas mais estradas para atender ao excessivo número de veículos.

1.1 Motivação

Nas últimas duas décadas, a população tem realizado exigências aos governos municipais e estaduais para que assumam responsabilidades e tomem providências quanto às políticas urbanas, em especial as que tratam dos serviços referentes à infraestrutura (CLEZAR, 2006). É preciso que o planejamento e os projetos saiam do papel e se concretizem, proporcionando, assim, soluções práticas e funcionais à população. Contudo, apenas esperar o poder público exercer a sua responsabilidade não é o bastante. É necessário uma maior mobilização dos cidadãos, que devem contribuir com seus pleitos.

Uma opção para a mobilização da população é a implementação do conceito de cidade inteligente. Esse termo é designado para os centros urbanos que utilizam Tecnologias da Informação e Comunicação (TICs) para possibilitar eficiência no planejamento, execução e manutenção dos serviços e infraestruturas, pensando nos interesses dos habitantes (HARRISON; DONNELLY, 2011). A intensa utilização de *smartphones* pelas pessoas pode auxiliar a interação entre população e uma gestão inteligente.

Segundo o Instituto Brasileiro de Geografia e Estatística, 190.461.847 brasileiros (94,6%) utilizam aparelhos móveis para o uso da internet (IBGE, 2010). O celular é uma alternativa não só para comunicação, mas também, para resolver inúmeros problemas do dia a dia, como solicitar serviços para locomoção através de aplicativos. Com o acesso à tecnologia, localização e georreferenciamento, recursos dispostos pelo aparelho do usuário, torna-se possível registrar e obter informações, que, nesse contexto, são fundamentais, pois permitem identificar onde os problemas estão localizados. Além disso, com o uso da internet é viável aumentar a aproximação entre o cidadão e a gestão da cidade, criando uma forma de relacionamento. Essa relação possibilita ao cidadão obter informações acerca dos serviços executados, evitando que eles precisem ir a postos de atendimento público em busca das mesmas (SCHAFFERS et al., 2011; CHOURABI et al., 2012).

Cada serviço de responsabilidade do poder público, estadual ou municipal tem geralmente um meio próprio para comunicar-se com o cidadão. Dentre eles, podem-se citar o atendimento pessoal, telefone, e-mail, formulário web e aplicativo de celular. Pode-se perceber que existem vários meios para o cidadão manter-se informado e também dar sugestões para a resolução de problemas urbanos. Além disso, é perceptível que as falhas existentes nas cidades, relacionadas a diferentes questões, como infraestrutura, segurança, lixo, meio ambiente, acessibilidade e mobilidade, dentre outras, são desconexas umas das outras. Ademais, não há, em geral, mecanismos para uniformizar o acesso e permitir o compartilhamento dessas informações entre sistemas distintos.

Nesse intuito, faz-se necessário o desenvolvimento de estratégias que possibilitem viabilizar o gerenciamento de dados. Assim, os gestores podem fazer o acompanhamento das sugestões e estudos, além de traçar metas para a melhoria das cidades. Isso deve

ocorrer de modo que favoreça a disponibilização de recursos de maneira fácil, com regras específicas e uniformes, possibilitando a interoperabilidade entre sistemas distintos, mas abstraindo detalhes de sua implementação. Uma das formas mais utilizadas para esse fim é a construção e disponibilização de uma interface onde ocorra a programação de aplicativos ou *Application Programming Interface* (API), termo da língua inglesa, amplamente utilizado na área de Tecnologia da Informação (TI).

1.2 Objetivos

Diante desse contexto, o trabalho proposto tem por objetivo contribuir com a gestão das cidades através da implementação de uma API para o gerenciamento de uma base de dados acerca dos relatos de problemas urbanos. Além disso, uniformizar o acesso às informações armazenadas, tendo em vista o acompanhamento do histórico de relatos e resoluções de problemas urbanos.

Com o intuito de atingir o objetivo geral, alguns objetivos especificados, como:

- Implementar uma API de acordo com o estilo arquitetural Representational State Transfer (REST), com a finalidade de realizar a integração com diversas aplicações;
- Realizar um estudo sobres as aplicações existentes acerca dos relatos de problemas urbanos, a fim de analisar o funcionamento, dando ênfase a maneira como é realizado os relatos dos problemas, assim como, entender como os mesmos estão dispostos na aplicação;
- Elaborar uma taxonomia, com intuito de organizar os problemas conforme suas respectivas categorias e disponibilizar essas informações, propenso a colaborar no contexto dos problemas urbanos;
- Contribuir para o gerenciamento dos problemas urbanos, assim como definir uma padronização para comunicação entre os cidadãos e os gestores das cidades através do uso de aplicações;
- Desenvolver uma documentação para facilitar o entendimento dos desenvolvedores sobre o funcionamento dessa API, possibilitando que outras aplicações possam consumir a mesma;
- Realizar a validação da API através de testes com aplicações web e *mobile*.

1.3 Metodologia

Para o desenvolvimento do trabalho foi realizada uma revisão bibliográfica sobre o assunto, com a pretensão de aprofundar o conhecimento acerca do processo de urba-

nização, problemas urbanos, planejamento urbano e cidades inteligentes, mas enfocando nos problemas urbanos. Além disso, através da pesquisa realizada sobre os trabalhos relacionados constatou a necessidade da produção de uma taxonomia, com a finalidade de separar os problemas urbanos em categorias, a fim de obter-se uma melhor organização e entendimento. Assim, elaborou-se uma taxonomia dividindo os problemas em sete categorias.

Dessa forma, tendo como base toda a pesquisa realizada e o levantamento de requisitos, utilizou-se a linguagem Javascript para a implementação da API. Ela foi executada na plataforma Node.js, com o auxílio do *framework* Express, que se conecta e faz interação com o banco de dados, sendo escolhido o ArangoDB, responsável por todo o armazenamento dos dados.

Com este trabalho, pretende-se que a API, através da documentação, possa ser de fácil utilização, sendo, portanto, empregada como base de dados para relatos de problemas, permitindo que esses sejam capazes de ser categorizados, no intuito de obter-se uma melhor representação.

1.4 Estrutura do documento

Para melhor explicar o desenvolvimento deste trabalho, sua divisão ocorre da seguinte maneira:

- A seção 2 faz uma revisão bibliográfica dos principais temas aqui abordados, como problemas urbanos e Arquitetura Orientada a Serviços, com ênfase na arquitetura REST, utilizada para o desenvolvimento deste estudo e dos trabalhos relacionados;
- A seção 3 apresenta a API proposta, mostrando a taxonomia desenvolvida com os problemas divididos em categorias. A modelagem da API tem como base o modelo orientado a grafos, diagrama de classe e objeto utilizados para uma melhor visualização das funcionalidades, e o banco de dados utilizado;
- A seção 4 descreve a implementação e validação com as tecnologias utilizadas para a implementação da API, expondo como estão estruturadas através de um diagrama de componente as respectivas rotas e, por fim, a validação da API sendo consumida por uma aplicação e *mobile*;
- A seção 5 expõe as considerações finais e os trabalhos futuros; e, no Apêndice, todas as rotas disponíveis.

2 Referencial Teórico

Nesta seção, serão discutidos conceitos acerca da Arquitetura Orientada a Serviços (SOA), como também, a elucidação de serviços, além das principais tecnologias e padrões utilizados para implementação do SOA, dando ênfase para o estilo arquitetural REST, a explanação de comparativos para um melhor entendimento e embasamento deste trabalho e, por fim, os trabalhos relacionados.

2.1 Arquitetura Orientada a Serviços

O desenvolvimento de aplicações não é uma tarefa fácil devido à complexidade dos softwares e o curto prazo de entrega determinado pelo cliente. Para sua construção envolve vários fatores como: conhecimento, dedicação, experiência dentre outros. Além disso, um alto custo (FOWLER, 2002). Conforme Pressman e Sommerville, durante o desenvolvimento é importante estabelecer uma arquitetura bem estruturada a fim de facilitar o processo de desenvolvimento, manutenção e minimizar os custos.

A arquitetura de software é um conceito abstrato, desse modo, existem várias definições. Segundo a ANSI/IEEE a arquitetura de um software apresenta os componentes fundamentais de um sistema e como seus componentes relacionam-se entre si (MAIER; EMERY; HILLIARD, 2004). Os estilos arquiteturais são formas de estruturar e auxiliar na projeção de softwares, funcionando como um modelo para organização dos componentes e dos módulos de uma aplicação. De acordo com Pressman (2005) e Sommerville (2011) existem diferentes estilos de arquitetura de software, dentre elas: arquitetura em camadas, modelo de repositórios, arquitetura orientada a objetos, arquitetura cliente servidor e arquitetura orientada a serviços.

A arquitetura orientada a serviços está sendo cada vez mais utilizada pelas empresas como uma boa estratégia para desenvolvimento de aplicações e para o aumento de produtividade. Esse sistema é fundamentado em princípios de computação distribuída. De acordo com Sampaio (2006) a *Service Oriented Architecture* (SOA) é uma arquitetura de desenvolvimento com a finalidade de criar um sistema modularizado. Esses módulos são funcionalidades nomeadas de serviços, com baixo acoplamento e que permitem a reutilização de código.

Para Stal (2006), as melhores implementações de SOA são realizadas quando, de fato, os desenvolvedores conseguem compreender esse paradigma como uma concepção de arquitetura. Assim, com esse conhecimento, facilitam-se as implementações e, consequentemente, conseguem um dos principais objetivos quando se adota a orientação a serviços,

que é a redução de dependências entre as funcionalidades. Além disso, para Furtado et al. (2009):

> O objetivo de SOA é permitir às organizações realizarem seus negócios e ter vantagens tecnológicas por meio da combinação de inovação de processos, governança eficaz e estratégia de tecnologia, as quais giram em torno da definição e reutilização de serviços. Um dos benefícios mais importantes que SOA fornece é a melhora da produtividade e agilidade tanto para o negócio quanto para TI, e conseqüentemente, a redução de custos no desenvolvimento e manutenção dos sistemas envolvidos (FURTADO et al., 2009, p. 7).

A proposta desse paradigma é proporcionar uma maior integração entre os sistemas. A Figura 1 representa as principais características dessa arquitetura, a qual está fundamentada em serviços, juntamente com as boas práticas dispostas aos usuários, a fim de ser utilizada para o desenvolvimento de aplicações. Em conformidade com Moraes (2011) uma nova maneira é utilizar processos de negócios que correspondem ao serviço ser uma parte do sistema, sendo, assim, uma alternativa ao uso de sistemas monolíticos. Já para Müller et al. (2008), a SOA dispõe de princípios e melhores práticas para implementação e execução de processos de negócios em ambientes heterogêneos.

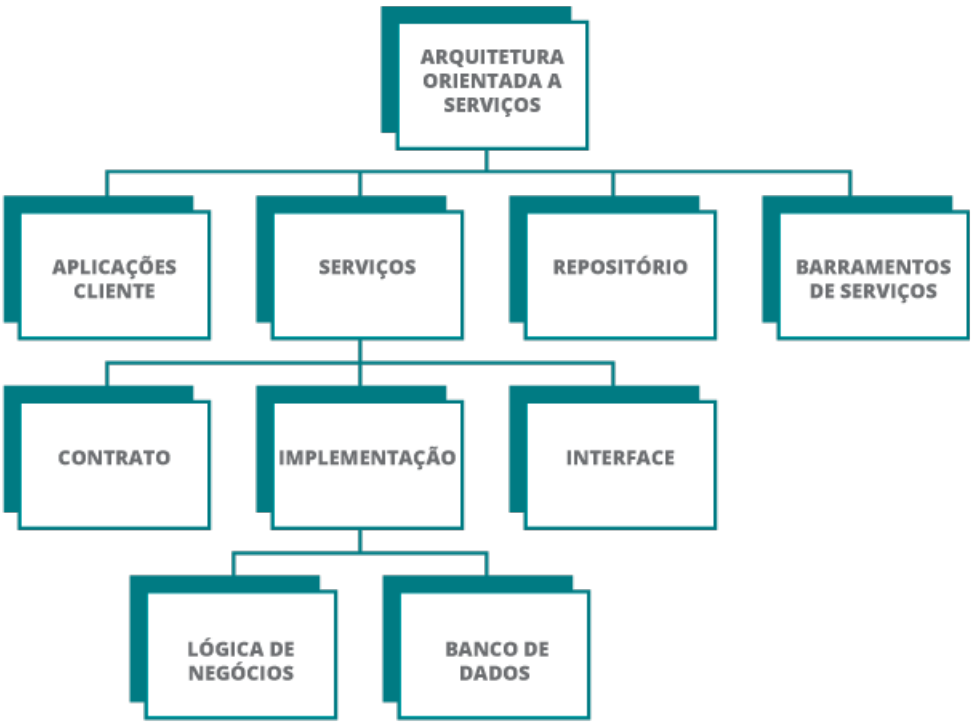
Figura 1 – Estrutura geral da arquitetura orientada a serviços.

Fonte – Adaptado de ADELIN (2019).

A implementação em cada empresa pode variar, porém é identificada pela introdução de novas tecnologias e plataformas que suportam especificamente o desenvolvimento, a execução e a evolução das soluções orientadas a serviços, podendo ser implementadas com a junção de várias tecnologias, APIs, produtos, extensões de infraestrutura, dentre outras partes (ERL, 2009).

A Figura 2 mostra os principais componentes da arquitetura orientada a serviços: os serviços, constituídos pelo contrato de acesso; implementação, interface que envolve abrange a lógica de negócios e acessos ao banco de dados; o repositório, onde ficam armazenados todos os serviços dispostos e o protocolo de troca de mensagens, permitindo o acesso aos serviços e às aplicações de clientes que utilizam os recursos disponibilizados através da arquitetura (ARAÚJO, 2011).

Figura 2 – Principais componentes da arquitetura orientada a serviços.



Fonte – Adaptado de ARAÚJO (2011).

2.1.1 Serviços

Para o entendimento do conceito de Arquitetura Orientada a Serviços, faz-se necessário o conhecimento da definição de serviço. Como um conceito geral, serviço é um ato ou efeito de servir (AURÉLIO, 2019). Conforme Barry (2003), um serviço pode ser software ou um hardware, enquanto software refere-se às funcionalidades que o sistema fornece.

Segundo Furtado et al. (2009) os serviços são funcionalidades ou módulos de negócio que possuem interfaces e são invocados através de mensagens, em que não se tem acesso ao serviço em si, mas apenas à sua interface. Outra característica, é que os serviços devem

ser independentes de outros, suas funcionalidades precisam ser bem definidas, ocorrendo a possibilidade de fazer composição entre eles.

Para Thomas Erl "A orientação a serviços é um paradigma que abrange um conjunto específico de oito princípios de design, onde a unidade mais fundamental da lógica orientada a serviços é o serviço" (ERL, 2009, p. 25). São eles:

- **Contrato de serviço padronizado**
  Os serviços apresentam o seu objetivo e capacidades por meio da contratação de serviços. "Um contrato de serviços pode ser composto de um grupo de documentos de descrição dos serviços, cada um dos quais descrevendo uma parte do serviço" (ERL, 2009, p. 76). No contrato do serviço, fica acordado a quantidade de conteúdo que será publicado, com a finalidade de definir as capacidades, modelo de dados e funcionalidades de uma determinada atividade. Além disso, como irão expressar granularidade e outras questões relacionadas à consistência, confiabilidade e governabilidade.

- **Baixo acoplamento de serviço**
  O baixo acoplamento de serviço diz respeito à independência entre contrato do serviço, implementação e os consumidores do serviço, isto é, representa o nível de dependência entre os serviços. Ainda segundo Erl (2009), essa característica faz com que um serviço seja capaz de progredir independentemente da sua implementação, garantindo a interoperabilidade, que, de acordo com o IEEE, consiste na competência de um sistema ou produto comunicar-se com outro sistema, garantindo ao usuário utilização dos serviços providos (IEEE, 2016).

- **Abstração**
  Esse princípio determina que um serviço possa ser genérico ao ponto de adaptar-se ao máximo às composições possíveis. Porém, ao mesmo tempo, não pode ser tão abstrato à medida que o consumidor não consiga saber do que se trata. Um contrato de serviço deve ser composto apenas por informações que sejam necessárias para os usuários do serviço. No entanto, os detalhes de implementação não são pertinentes no contrato. Durante a fase de implementação, o nível do serviço deve ser planejado. O excesso de informações pode causar o uso inadequado de serviço, podendo resultar em problemas de acoplamento. Assim afirma Erl (2009):

  > Como esse princípio resulta na ocultação deliberada de informações, precisamos determinar cuidadosamente que informações devem ser expostas. Cada parte dos metadados disponível pode ser utilizada de um modo que pode ter consequências inesperadas no futuro (ERL, 2009, p. 140).

- **Capacidade de reuso de serviço**
  A capacidade de reuso de serviço é considerado um dos princípios mais relevantes

para a implementação de uma arquitetura orientada a serviços. Segundo Erl (2009) para que o serviço disponha da capacidade de ser reutilizado recomenda-se que seja implementado de maneira genérica e agnóstico. Assim, conforme o mesmo autor, "Um serviço é agnóstico quando sua lógica é independente dos processos de negócio e da plataforma tecnológica proprietária ou de aplicativos proprietários". (ERL, 2009, p. 155).

Para Oliveira, Ramos e Junior (2014) o principal motivo para que o reuso de serviço esteja sendo muito utilizado em grandes empresas e negócios é por ser uma maneira para alcançar a flexibilidade, a interação e possibilitar a redução de custos no desenvolvimento de softwares. Além disso, em consonância com Pelechano et al. (2011) outro aspecto importante é viabilização da construção de novos produtos, propiciando, assim, o aumento da agilidade de entrega.

- **Autonomia de serviço**
  Esse princípio assegura que os serviços possam executar sua lógica independentemente de fatos externos ou mesmo de outros serviços. Está relacionado com baixo acoplamento, pois quanto mais recursos compartilhados utilizar, menor será a sua autonomia do serviço. Dessa maneira, Pereira (2012), afirma que quanto maior a autonomia do serviço, melhor para alcançar confiabilidade e previsibilidade dos programas de software.

- **Independência de estado do serviço**
  O estado do serviço refere-se à condição atual do mesmo. O objetivo desse princípio é assegurar um melhor desempenho através do isolamento da responsabilidade de guardar-se do estado. Ao receber uma requisição o serviço deve tratá-la e respondê-la conforme solicitado. Para Erl (2009), esse princípio é destinado com o intuito de diminuir o consumo de recursos que surgem do processamento desnecessário do gerenciamento de estado.

- **Visibilidade do serviço**
  No momento que um recurso ou serviço não se encontra devidamente visível, normalmente, os usuários têm dificuldade de usá-lo e acabam desenvolvendo um recurso próprio, podendo causar duplicação de um recurso já existente (PEREIRA, 2012, p. 28). Para uma fácil utilização pelos usuários, o serviço deve ser de fácil interpretação e descoberta. Logo, necessita ser genérico a fim de atender a diversas causas ao padronizar o serviço em um dado ambiente e, assim, propiciar a descoberta de serviços. Após o usuário encontrar um serviço, caso o consumidor consiga detectar as funcionalidades providas, é possível afirmar que o serviço pode ser interpretado.

- **Composição de serviços**
  Os serviços devem ser projetados dispostos a composições e decomposições, ou seja, um serviço pode ser dividido em partes menores a fim de ter várias soluções para

resolver problemas diferentes. A composição pode ser considerada como uma forma de reuso. Uma composição requer pelo menos dois serviços. Acerca disso, Erl (2009), afirma que:

> À medida que a sofisticação das soluções orientadas a serviços continua aumentando, aumenta também a complexidade das configurações de composição do serviço subjacentes. A capacidade de compor efetivamente os serviços é um requisito crucial para alcançar alguns dos objetivos mais fundamentais da computação orientada a serviços (ERL, 2009, p. 46).

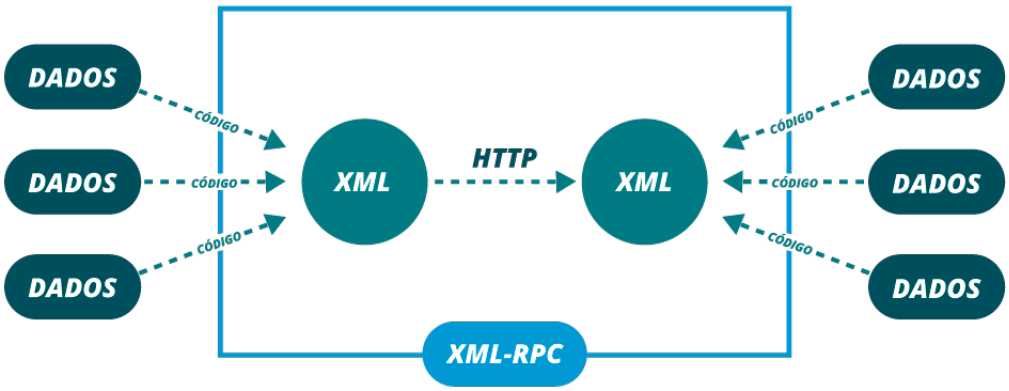
2.2 Principais tecnologias aplicadas a SOA

A arquitetura orientada a serviço é um modelo genérico, existem várias formas de implementar os serviços, contratos e a comunicação entre os componentes. A seguir serão expostas algumas tecnologias e padrões que podem ser utilizados para o desenvolvimento de aplicações com base nessa arquitetura.

2.2.1 XML-RPC

A tecnologia XML-RPC foi desenvolvida no final da década de 90 por Dave Winer. Segundo Laurent et al. (2001) é um protocolo de chamada de procedimento remoto (RPC) que utiliza como representação de dados o XML para codificar suas chamadas e o HTTP como mecanismo de transporte, como é possível observar na Figura 3.

Figura 3 – Exemplo de XML-RPC.



Fonte – Adaptado de (STERVINOU, 2019).

Por meio do RPC, é possível realizar uma chamada de um procedimento que não se encontra implementado localmente. O código pode ser escrito em linguagens de programação diferentes. Além disso, facilita o desenvolvimento de aplicações distribuídas, pois não necessita aprender sobre protocolos subjacentes, redes e vários detalhes de implementação (PULUCENO, 2012).

2.2.2 Web Services

A necessidade de integração de sistemas e a padronização das comunicações entre plataformas distintas como Windows, Mac, Linux, dentre outras, assim como as linguagens de programação, provocou o surgimento do *Web Services* no final da década de 90. Esse sistema representou uma inovação diante de outros modelos de computação distribuída existentes como CORBA, RMI e DCOM, daquela época, que não obtiveram tanto êxito.

Desenvolvida pelo World Wide Web (W3C), a tecnologia *Web Services* é definida como um sistema de software criado para suportar a interoperabilidade entre máquinas na rede (W3C, 2019). Com essa tecnologia torna-se possível a interação de sistemas desenvolvidos em plataformas diferentes.

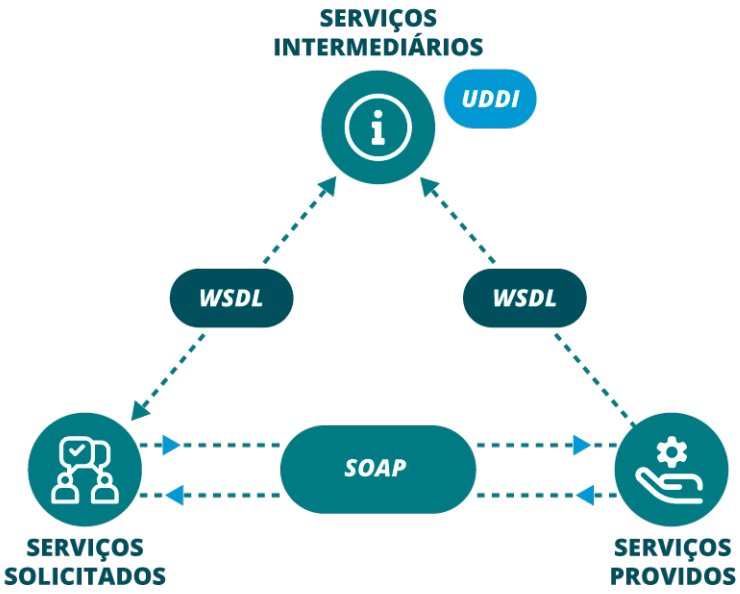
O funcionamento da *Web Services* tem como base os padrões eXtensible Markup Language (XML) e Simple Object Access Protocol (SOAP), como mostra a Figura 4. O transporte dos dados é realizado através do protocolo HTTP ou HTTPS. Os dados com operações, mensagens, parâmetros, dentre outros, são transferidos no formato XML, os quais são encapsulados pelo protocolo SOAP e descritos através da linguagem *Web Services Description Language* (WSDL). O protocolo *Universal Description, Discovery and Integration* (UDDI) é utilizado para os processos de publicação, pesquisa e descoberta de Web Services (CRUZ, 2010).

De acordo com CRUZ (2010), *Web Services* são várias tecnologias necessárias para o SOA e que tem sido muito utilizadas para o desenvolvimento de aplicações por obterem respostas positivas, adaptáveis e flexíveis. Segundo Gomes (2014) existem dois padrões para o desenvolvimento de Web Services: o padrão SOAP e o REST ou RESTful, utilizado neste trabalho.

2.2.3 Representational State Transfer (REST)

A procura de qualidade, desempenho e manutenção dos serviços remotos impulsionou o surgimento da *Representational State Transfer*, em tradução livre Transferência de Estado Representacional. Essa surgiu no início dos anos 2000, a partir da tese de doutorado de Roy Fielding, da University Of California, “Architectural Styles and the Design of Network-based Software Architectures” (Estilos arquiteturais e o Design de Arquiteturas de Software Baseados em Redes), que mostra a análise de diversas arquiteturas.

Figura 4 – As bases para construção de um Web Services.



Fonte – Adaptado de CRUZ (2010).

Assim, Fielding e Taylor (2000) define REST como estilo arquitetural que reúne uma série de princípios e restrições para o desenvolvimento de sistemas distribuídos e hipermídias. A composição desse estilo arquitetural surge de um estilo nomeado por Fielding como "Desconhecido", mas que ao adicionar características passa a ser chamada de restrições REST. Sob esse ponto de vista, Fielding e Taylor (2000) afirma que:

> Examinando o impacto de cada restrição, como isso é adicionado ao estilo referenciado, pode-se identificar as propriedades induzidas pelas limitações da Web. Restrições adicionais podem então ser aplicadas para formar um novo estilo arquitetural que melhor reflita as propriedades desejadas de uma arquitetura Web (FIELDING; TAYLOR, 2000, p. 76).

Dessa maneira, o objetivo geral é a padronização do conjunto de melhores práticas denominadas *constraints*, com o intuito de determinar a forma na qual padrões como *Hypertext Transfer Protocol* (HTTP) e *Uniform Resource Identifier* (URI) devem ser modelados, a fim de aproveitar dos recursos oferecidos pelos mesmos. Existem seis restrições que foram pensadas para constituir o estilo arquitetural. As restrições (*constraints*) do REST são:

I. **Cliente-Servidor:** por meio dessa restrição torna-se possível separar as responsabilidades de diferentes partes do sistema. Essa divisão pode ocorrer de diversas formas, um exemplo é a separação de dados do *front-end* da aplicação. Há várias ferramentas que atualmente seguem esse padrão, como os *frameworks*, o AngularJS e a APIs

RESTful, possibilitando o isolamento de forma organizada para cada uma dessas funcionalidades (DIAS, 2016, p. 13).
De acordo com Fielding e Taylor (2000):

> Ao separar as preocupações de interface de usuário das preocupações de armazenamento de dados, podemos melhorar a portabilidade da interface do usuário em diversas plataformas e elevar a escalabilidade por meio da simplificação dos componentes do servidor (FIELDING; TAYLOR, 2000, p. 45).

O foco na portabilidade de interfaces em diferentes plataformas, alicerçada na arquitetura cliente e servidor, que é fundamentada na definição de *Web Services*, apresenta o "write once, run anywhere" (escreva uma vez, execute em qualquer lugar). A camada de negócio desenvolvida no servidor, disponibiliza via HTTP, possibilita que qualquer cliente sendo ele web, desktop ou móvel, seja capaz de realizar a comunicação com essa camada (RIBEIRO; FRANCISCO, 2016).

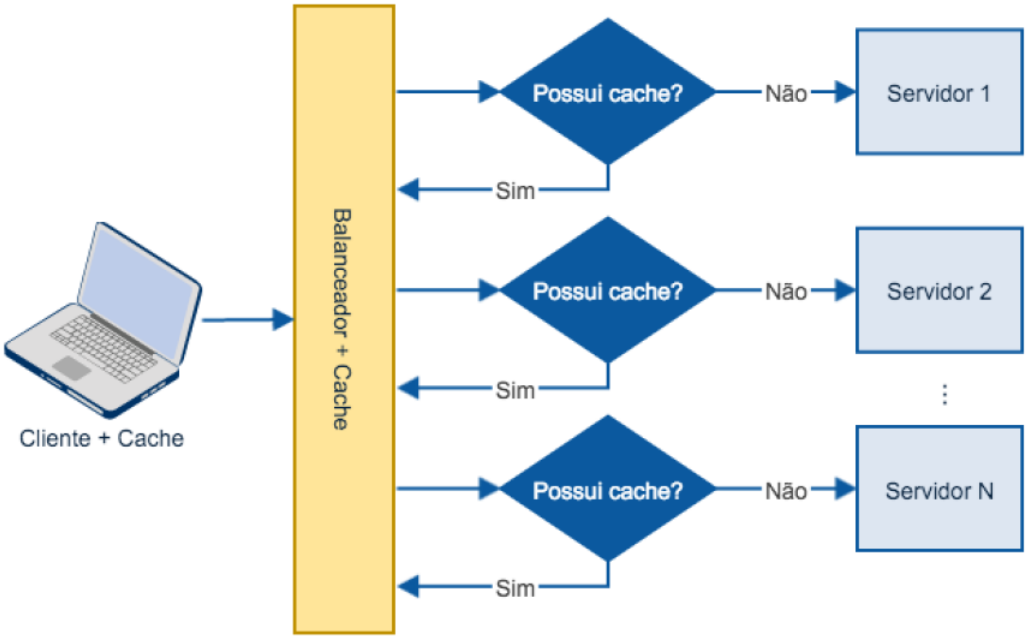
II. **Stateless:** significa um protocolo sem estado. Propõe que as requisições sejam independentes, ou seja, que cada requisição ao servidor não tenha ligação com nenhuma outra. Desse modo, as requisições devem abranger o mínimo de informações possíveis, as necessárias para que sejam analisadas com sucesso pelo servidor. Conforme Fielding e Taylor (2000) "cada pedido do cliente para o servidor deve conter todas as informações necessárias para compreender o pedido" (FIELDING; TAYLOR, 2000, p. 79).

III. **Cache:** para obtenção de um maior desempenho na arquitetura REST as respostas devem permitir o uso de cache. Os navegadores controlam bem o uso de cache, além de permitir o armazenamento das respostas. O HTTP possibilita o funcionamento adequado dessa restrição a partir da inserção de cabeçalhos "expires" na versão 1.0 do HTTP ou como opção na versão 1.1 do HTML, usando o "cachecontrol" (FERREIRA; KNOP, 2017).

   As solicitações HTTP realizadas pelo navegador são antes encaminhadas ao cache do mesmo, verificando se há uma resposta válida armazenada no cache para atender à solicitação, a melhor é aquela que não precisa comunicar-se com o servidor. Assim, se houver uma resposta será lida do cache, permitindo consequentemente eliminar toda a latência de rede e evitar cargas de dados para a transferência de dados (GRIGORIK, 2019).

   Cada resposta às requisições deve informar para os clientes ou elementos intermediários como: *proxies, gateways* e/ou balanceadores de carga qual, a política de cache mais adequada. Percebe-se na Figura 5 situações nas quais o cliente local ou o balanceador possui cache. Dessa forma, não tem a necessidade de requisitar o processamento e uma nova requisição ao servidor. Portanto, esse modelo permite ao servidor executar somente as tarefas que são necessárias.

Figura 5 – Representação da restrição cache.



Fonte – Adaptado de Dias (2016).

IV. **Interface Uniforme:** essa é a principal característica da REST que a diferencia das outras arquiteturas. Desenvolver uma aplicação com interface uniforme tem como objetivo simplicidade, acessibilidade, interoperabilidade e a capacidade de descoberta de recursos. (COSTA, 2014).

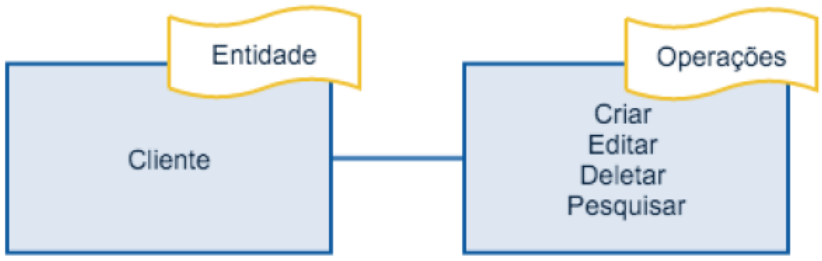
Para obtenção de uma interface uniforme é necessário quatro requisitos:

- **Identificação de Recursos:** os serviços da Arquitetura REST são baseados nos recursos que podem ser definidos como elementos de informações, responsáveis pela manipulação através de endereços, os URI, que realizam os endereçamentos dos recursos.
- **Representação de Recursos:** os recursos são manipulados por meio de representações, que podem ser em JavaScript Object Notation (JSON), XML, HTML, dentre outras.
- **Mensagens Autoexplicativas:** para o entendimento da requisição ou resposta do que foi enviado pelo cliente ou servidor é preciso a passagem de metadados. Os mais importantes metadados para o HTTP são o código, o método HTTP, host, media type, idioma e a compressão utilizada, dando destaque aos cabeçalhos do HTTP implementados em APIs REST (OLIVEIRA, 2018).
- **Hypermedia as The Engine of Application State (HATEOAS):** a Hi-

permídia como Motor de Estado de Aplicativos opera como um motor de navegação implementado de forma dinâmica e acrescentado em representações fornecidas pelo Web Service. De acordo com Cavaleiro (2013) o princípio do HATEOAS define como uma aplicação cliente pode comunicar-se com outras aplicações em rede através de hipermídias. Além dos dados solicitados, as respostas das APIs REST devem ter uma requisição, hyperlinks com as informações sobre os recursos disponíveis da API.

Para compreender mais acerca do que é interface, o autor Dias (2016), em seu trabalho intitulado "Desmitificando REST com JAVA", cita como exemplo a manipulação de vendas em um *e-commerce*, por uma loja virtual, onde existem diversos componentes como: produto, cliente, pedido, entre outros. Desse modo, deve-se pensar em criar uma interface que permita a utilização desses conceitos. Na Figura 6 há um exemplo utilizado pelo autor:

Figura 6 – Representação de uma interface.



Fonte – Adaptado de Dias (2016).

Veja que a Figura 6 representa o recurso "cliente" e uma série de funcionalidades que podem ser executadas sob o mesmo. Em seguida, será mostrado uma API proposta por este trabalho, com a implementação dessa interface, a qual cada uma das operações representa um método do protocolo HTTP.

- **Sistemas em Camadas:** consiste na utilização de camadas com objetivo de separar as unidades diferentes das funcionalidades de uma aplicação, objetivando expandir o desenvolvimento de aplicações distribuídas, em que uma API REST necessita conter camadas. Além disso, deve conseguir inserir elementos intermediários que não sejam visualizados pelo cliente a fim de obter a escalabilidade.

  Desse modo, em conformidade com Fielding e Taylor (2000) um sistema em camadas permite que a arquitetura, constituindo-se por diversos níveis hierárquicos, consiga restringir os componentes participantes de maneira que não

seja capaz de enxergar através da camada adjacente. Assim, o conhecimento do sistema que o componente possui fica restrito a camada apenas que faz parte.

- **Código sob Demanda:** essa restrição é opcional. Segundo Fielding e Taylor (2000) em sua tese, a capacidade de expansão de um software sem a necessidade de ocorrer a quebra do mesmo denomina-se extensibilidade. Ao utilizar um código sob demanda pode-se adaptar o cliente sem interromper o sistema para adquirir novas funcionalidades e requisitos. Tal restrição é permitida quando o cliente consegue expandir a parte da lógica do servidor com os seus próprios serviços (FERREIRA; KNOP, 2017).

2.2.4 Hypertext Transfer Protocol (HTTP)

O Hypertext Transfer Protocol surgiu em 1996 por meio de um trabalho realizado em conjunto por Roy Fielding e Henrik Frystyk Nielsen, sendo até hoje utilizado. O HTTP refere-se a um protocolo da camada de aplicação, segundo o modelo o Open System Interconnection (OSI). De acordo com Kurose e Ross (2013) é executado em dois programas: um cliente e outro servidor, com a finalidade de facilitar a manipulação nas aplicações. Esse protocolo é responsável por fazer a transferência de recursos através da Internet, em que os métodos HTTP permitem a interação com os recursos, criando uma interface uniforme que pode ser acessada pelos inúmeros dispositivos.

O protocolo HTTP é um modelo simples, resume-se basicamente em requisições (*request*), ocorrem do lado do cliente, e respostas (*response*), acontecem do lado do servidor. Elas seguem os padrões definidos pelo protocolo, assim, para realizar requisições é preciso estabelecer conexão entre o cliente e servidor (SILVA; MONTE-MÓR, 2016).

Segundo Tobaldini (2007) na primeira versão do protocolo havia apenas o mecanismo de busca de recursos (GET), além disso, não suportava especificações de versão de protocolo e cabeçalhos de informação nas trocas de mensagem. Desta maneira, a falta de um método de envio de informações do cliente para o servidor limitava as funcionalidades. Já a versão HTTP 1.1 possui oito métodos que são: GET, POST, PUT, DELETE, OPTIONS, HEAD, TRACE e CONNECT.

Os serviços Web REST baseiam-se no protocolo HTTP e utilizam seus principais métodos POST, GET, PUT e DELETE para operar sobre os recursos. Cada método está relacionado a uma função, com exceção do PUT que pode ser usado por mais de uma maneira. Os métodos podem ser executados sobre os recursos através do seu URI.

- **POST:** utilizado para criar um novo recurso através de uma representação como resposta. Nele o servidor envia ao cliente um código que informa a respeito do sucesso ou falha da operação. Logo, é possível enviar outra informação como uma mensagem

de erro, no caso de falha, ou um dado de localização, que informará onde está situado o novo recurso recém-criado;

- **GET:** recupera dados dos recursos. Não é utilizado para executar ações, mas apenas para retornar dados. É considerado idempotente, isto é, independentemente da quantidade de vezes que é executado sob um recurso o resultado não altera. Ao inserir um endereço no navegador web realiza-se uma requisição GET junto ao servidor com o objetivo de receber a representação para aquela Universal Resource Locator (URL);
- **PUT:** usado para fazer atualização de um determinado recurso. Em alguns contextos, muitos específicos, pode ser utilizado como uma forma de criação;
- **DELETE:** tem como finalidade a remoção de um recurso. Caso um recurso inexistente for requisitado para deleção é retornado ao status de erro 204.

Nota-se que ao realizar uma requisição para o servidor é retornado um código de status, um número de três dígitos, que resume a resposta enviada pelo servidor e auxilia aos clientes na interpretação. Esses códigos são divididos em cinco famílias: 1xx, 2xx, 3xx, 4xx e 5xx, sendo:

- 1XX - Informacionais;
- 2XX - Códigos de sucesso;
- 3XX - Códigos de redirecionamento;
- 4XX - Erros causados pelo cliente;
- 5XX - Erros originados no servidor.

A Tabela 1 mostra os códigos de Status HTTP organizados por famílias:

Tabela 1 – Status HTTP.

Família	Lista de Código de Estado HTTP	Descrição
1xx	100	Continue
	101	Switching Protocol
2xx	200	OK
	201	Created
	202	Accepted
	203	Non-Authoritative Information
	204	No Content
	205	Reset Content
	206	Partial Content
3xx	300	Multiple Choices
	301	Move Permanently
	302	Found
	303	See Other
	304	Not Modified
	305	Use Proxy
	307	Temporary Redirect
4xx	400	Bad Request
	401	Unauthorized
	402	Payment Required
	403	Forbidden
	404	Not Found
	405	Method Not Allowed
	406	Not Acceptable
	407	Proxy Authentication Required
	408	Request Time-out
	409	Conflict
	410	Gone
	411	Lenght Required
	412	Precondition Failed
	413	Request Entity Too Large
	414	Request-URI Too Large
	415	Unsupported Media Type
	416	Requested range not satisfiable
	417	Expectation Failed
5xx	500	Internal Server Error
	501	Not Implemented
	502	Bad Gateway
	503	Service Unavailable
	504	Gateway Time-out
	505	HTTP Version not supported

Fonte – Adaptado de (RUNSCOPE, 2019).

Em consonância com Matos (2013) um dos conceitos mais importantes na arquitetura REST são recursos utilizados para identificar de forma única um objeto abstrato ou físico, que substituem a ideia dos métodos ou serviços utilizados em RPC e serviços Web SOAP. Por meio da existência dos recursos pode-se tornar a arquitetura REST simples. Os recursos relacionam-se com outros recursos, e com os métodos que operam sobre ele. Para manipulação dos recursos, os componentes da rede comunicam-se geralmente através de HTTP. Os métodos que atuam sobre os recursos correspondem aos métodos padrões do HTTP: GET, POST, PUT e DELETE. Os recursos são descritos por URLs lógicos, designados nesse contexto por URIs.

Os URIs tornam os recursos endereçáveis por todos os outros componentes do sistema e identificam um recurso. Além disso, possuem meios de agir sobre esses ou de obter a sua representação pela descrição do seu mecanismo de acesso primário ou sua localização na rede. A princípio cada coleção e recurso de uma API dispõe do seu próprio URI (MATOS, 2013).

A Universal Resource Locator pode ser usada para fornecer caminhos, sendo uma forma de URI. Para entender melhor esse processo, como exemplo, pode-se observar o caso da listagem de cidadãos, em que é possível decompor a URL utilizada para localização da listagem em várias partes:

http://localhost:8000/reporte/cidadao

- http:// - informa qual protocolo está sendo utilizado, nesse caso, o HTTP;
- localhost:8000 - indica o servidor de rede que está sendo utilizado e a porta não é especificada, trata-se da padrão no caso do protocolo HTTP 80;
- reporte - trata do contexto da aplicação, ou seja, a raiz a qual a aplicação está sendo fornecida para ser consumida.
- cidadao - é o endereço do recurso, no caso, a listagem de cidadao.

2.2.5 Formato de Dados

O formato de dados permite a solicitação e o recebimento de informações através de *Web Services*. No ambiente das APIs, as informações são propagadas nos formatos mais adequados para serem utilizadas por diferentes aplicações. Os modelos mais utilizados na comunicação web são XML e JSON (RICHARDSON, 2013).

O formato Extended Markup Language ou XML é recomendado para criação de documentos organizados. De acordo com Lima e Carvalho (2005) é uma metalinguagem que define uma sintaxe podendo ser utilizada na criação de outras linguagens de marcação, com estruturas e semântica própria. Além disso, particiona os documentos, constituindo

um conjunto de regras para a definição de marcadores semânticos. O Código 1 mostra uma estrutura de documento XML:

Código 1 – Exemplo de XML.

```
<?xml version="1.0" encoding="UTF-8"?>
<reporteCidadao>
  <cidadao>
    <cidadao nome="Elisa Andrade" email="elisandradecc@gmail.com" sexo="
   F">
    <cidadao nome=" Leonardo Bandeira" email="leonardolucena.cc@gmail.
   com" sexo="M">
     </cidadao>
</reporteCidadao>
```

Fonte – Autoria própria.

O XML é bastante utilizado pelos desenvolvedores e por diversas plataformas tecnológicas como Oracle, .NET e Java. De acordo com o World Wide Web (W3C) a linguagem XML apresenta as seguintes características: ser diretamente útil na Internet; ser compreensível; facilitar a publicação eletrônica; permitir protocolos para troca de dados pelas empresas independentemente da plataforma de hardware e software; simplificar para os usuários o processamento de dados pelo uso de aplicações de baixo custo; viabilizar a utilização de metadados ajudando na busca de informações e aproximar fornecedores e consumidores de informação.

Já o formato de dados JSON foi desenvolvido por Douglas Crockford, nativo da linguagem JavaScript. Conforme a RFC 4627, JSON é um formato de dados baseado em texto, com a finalidade de intercâmbio de dados independente da linguagem de programação. Veja a seguir um exemplo:

Código 2 – Exemplo de JSON.

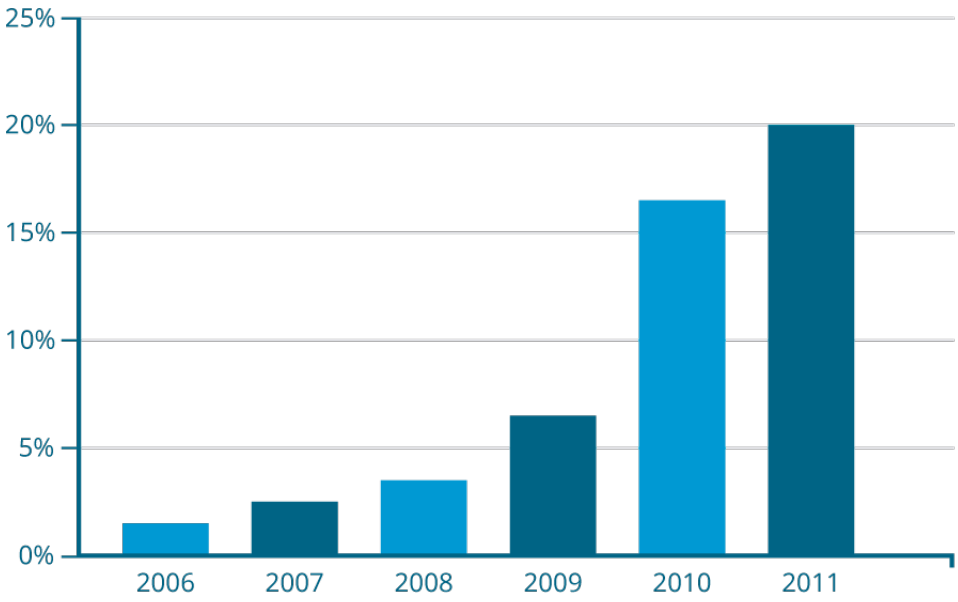
```
{
    "cidadao":{
    "nome": "Elisa Andrade",
    "email": "elisandradecc@gmail.com"
    }
}
```

Fonte – Autoria própria.

O JSON tem uma sintaxe simples, desse modo, seu processamento é mais eficiente comparado ao XML. Além de apresentar uma facilidade de desenvolvimento, arquivo com tamanho reduzido e com alto de desempenho nas aplicações, que milhares de Web Services utilizam.

De acordo com DuVander (2011) em seu artigo, 1 in 5 APIs Say “Bye XML”, o XML estava presente em mais de 90% das aplicações no ano de 2009. No entanto, com a popularização da linguagem Javascript acabou perdendo espaço. As APIs implementadas utilizam mais o JSON do que o XML, como é possível observar acima, no Gráfico 7, que exibe o percentual das APIs utilizando apenas JSON no período 2006 a 2011.

Figura 7 – As APIs implementadas com JSON.



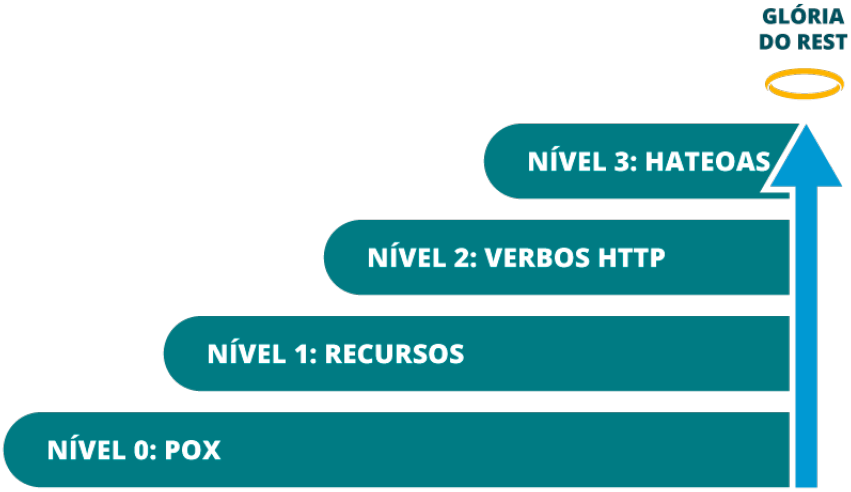
Fonte – (DUVANDER, 2011).

2.2.6 REST X RESTful

Anteriormente, foram vistos os conceitos e as características da arquitetura REST. Para uma API ser considera RESTful deve ter absolutamente as regras definidas na arquitetura REST e implantar o modelo Maturidade Richardson (2013), representado na Figura 8. O arquétipo possui 4 níveis, sendo 0, 1 e 2, considerados os mais fáceis de serem implementados.

- Nível 0 - Serviço de Transporte: as operações no servidor utilizam o protocolo HTTP como mecanismo de transporte. O POX é uma maneira de programar sem ter a preocupação com a definição dos recursos e o uso correto dos métodos e dos códigos de estado.

Figura 8 – Modelo de Maturidade de Richardson.



Fonte – Adaptado de (RICHARDSON, 2013).

- Nível 1 - Recursos: a API é modelada conforme os recursos disponíveis. Segundo Fowler (2002) esse nível possibilita que a interação ocorra com os recursos individuais disponíveis por meio da URI de cada recurso.
- Nível 2 - Verbos HTTP: consiste na implementação dos métodos mais utilizados (GET, POST, PUT e DELETE) para os diferentes tipos de operações CRUD (Create, Read, Update e Delete).
- Nível 3 - HATEOAS: controle de hipermídias. É o uso do HATEOAS que possibilita ao cliente uma maior interação com a aplicação, sem ter o conhecimento da documentação. Segundo Fielding e Taylor (2000) APIs que não utilizam HATEOAS não podem ser consideradas RESTful

2.2.7 XML-RPC x RESTful

Quando comparado o XML-RPC aos protocolos RESTful, que as representações de recurso são transferidas, percebe-se que o XML-RPC é projetado para chamar métodos, sendo mais estruturado, ou seja, o código da biblioteca comum pode ser usado para implementar clientes e servidores. Logo, há menos trabalho com design e documentação para um protocolo de aplicativo específico. Uma outra diferença entre esses protocolos, é que o RESTful utiliza o URI HTTP para informações de parâmetro, enquanto o XML-RPC usa o URI para identificar o servidor.

2.2.8 REST x SOAP

O protocolo SOAP surgiu como uma forma de realizar chamadas de procedimento remoto via HTTP, usando o também o XML como formato de dados. De acordo com Josuttis (2007) o REST é uma boa opção para implementação de Web Services, pois está em expansão, sendo, assim, considerada uma forma mais leve de integração e de fácil aprendizagem. Ademais, é uma das arquiteturas mais populares para o desenvolvimento de serviços distribuídos, facilitando a implementação de APIs para a comunicação entre os diferentes componentes do sistema. O SOAP é o modelo mais tradicional usado por grandes organizações, além disso, é um padrão recomendado pelo W3C.

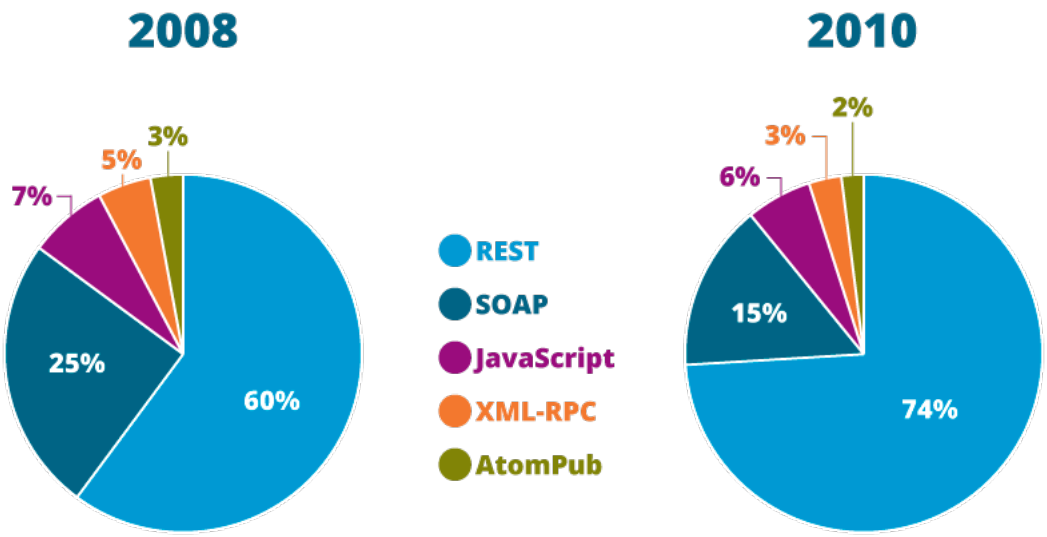
O serviço na arquitetura SOAP é orientado a ações, enquanto na REST é orientado aos recursos, os quais as ações (verbos HTTP) manipulam de modo direto os recursos. O REST pode utilizar diferentes tipos de formatos de dados, contudo, deve ser definido conforme a linguagem/aplicação desenvolvida enquanto o SOAP está limitado ao XML como formato de resposta. Ainda assim o SOAP é a linguagem mais fácil de ser interpretada, uma vez que os dados estão armazenados em elementos XML. Já o documento XML, é mais pesado devido à quantidade de informação a ser transferida entre os sistemas, requisitando uma maior capacidade de processamento o que, portanto, exige mais tempo (MARQUES, 2018).

Por meio de uma pesquisa realizada por Musser (2011) em maio de 2010, no mercado sobre a APIs Web, com 2000 APIs Web listadas no serviço de diretórios Programmable Web, concluiu-se, como aponta a Figura 9, que cerca de 1.500 dos serviços usam REST, enquanto apenas 360 usam SOAP.

Para obter-se uma visão mais atual sobre os serviços com maior utilização no desenvolvimento de APIs pode-se utilizar a ferramenta Google Trends, O Gráfico 10, que apresenta uma comparação das buscas entre os termos “api rest” e “api soap”. O REST é visto como a escolha predominante no mercado de aplicações Web considerando o intervalo de tempo entre janeiro de 2004 até janeiro de 2019.

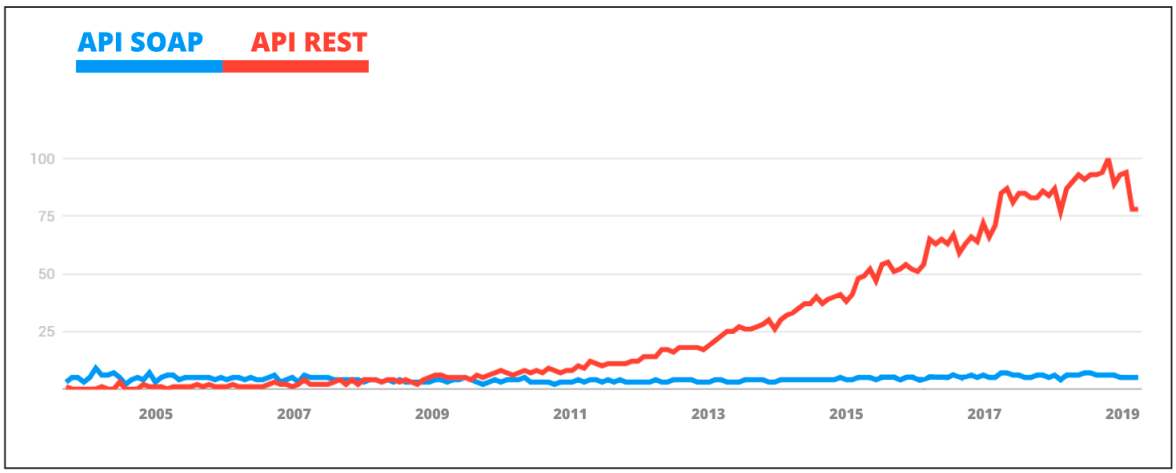
De acordo com Marques (2018) conclui-se que a REST é uma ótima alternativa para casos com limitações de recursos e de largura de banda. Além disso, é flexível, pois não apresenta restrições no formato em que a informação retorna, mas no comportamento dos componentes. O SOAP é indicado como a melhor solução para organizações com padrões rígidos e ambientes complexos onde a integridade, segurança e confiabilidade dos dados venham a ser fatores necessários na transferência dos mesmos.

Figura 9 – O uso do estilo REST no desenvolvimento de APIs.



Fonte – (MUSSER, 2011).

Figura 10 – Comparativo entre os termos "api rest"e "api soap".



Fonte – (GOOGLE, 2019a).

2.3 Trabalhos Relacionados

Esta subseção apresenta soluções no contexto do trabalho proposto. Assim, para uma melhor exposição, os trabalhos relacionados estão subdivididos em duas partes: trabalhos que desenvolveram APIs com base no estilo arquitetural REST; e aplicações brasileiras utilizadas para auxiliar na gestão das cidades, por meio do uso de aplicativos que realizem relatos de problemas urbanos.

O trabalho desenvolvido por Marques (2018) está inserido no contexto de desenvolvimento Web com base na arquitetura REST. Ele utilizou o Python como linguagem de programação e framework Django REST, utilizando o padrão Model-View-Controller (MVC) com a finalidade de desenvolver uma API REST para aplicação *cloud* em modelo de Software as a Service (Saas). Com isso, Marques (2018), visava a integração de uma aplicação Web com um Software de Controle Estatístico e Otimização de Processos. Além disso, realizar a recolha e armazenamento de dados sobre pesagens e processos de produção de produtos, de pré-embalados, para tratamento estatístico como cálculos estatísticos, elaboração de histogramas, cartas de controlo de Shewhart, entre outros.

Ferreira (2017), na sua monografia, desenvolveu um serviço de gerência de processos através do modelo arquitetural REST. Assim, a API implementada possibilitava criar novos simuladores, sem estar vinculado a uma arquitetura específica, utilizados para realizar a simulação de algoritmos como FIFO, Shortest Job First, dentre outros, para fins de aprendizado de conceitos da disciplina Sistemas Operacionais. Para o desenvolvimento da API, Ferreira (2017) utilizou a linguagem de programação JAVA, bem como, os padrões de projetos Strategy e Abstract Factory.

2.3.1 Aplicativos Brasileiros

Nome: Problemas Urbanos[1]
Desenvolvedor: Nuno Corte
Versão: 0.0.5 (5 de janeiro de 2017)
Plataforma: Android 4.1 ou superior
Avaliação: 4,8 (8 avaliações)
Descrição do Aplicativo: o aplicativo permite aos órgãos governamentais como a prefeitura, polícia, bombeiros e outros ter acesso às reclamações realizadas pela população no geral. Trata-se de uma forma de centralizar informações em que os problemas são armazenados em uma base de dados, podendo ser consultados em smartphones ou tablets com acesso à internet. Funciona por meio de contribuições informadas pela comunidade e é desenvolvido pela Nuno Corte, disponível para Android na Google Play.

[1] (GOOGLE, 2019c)

Nome: Pelas Ruas[2]
Desenvolvedor: Grupo RBS
Versão: 1.1.2
Atualizado: 5 de janeiro de 2017
Plataforma: Android 4.1 ou superior
Avaliação: 2,7 (184 avaliações)
Descrição do aplicativo: é um aplicativo com a finalidade de contribuir na gestão da cidade de Porto Alegre e regiões metropolitanas. Pelas Ruas apresenta uma maneira colaborativa para realizar-se o relato dos problemas urbanos. Possibilita a discussão e o compartilhamento das dificuldades encontradas nas regiões, além disso, a interação entre os usuários é feita por meio de postagens, tornando-o uma aplicação colaborativa. É oferecido pelo Grupo RBS no Google Play.

Nome: SP156[3]
Desenvolvedor: Metasix Tecnologia
Versão: 2.1.4
Atualizado: 14 de novembro de 2018
Plataforma: Android 4.1 ou superior
Avaliação: 2,8 (1.557 avaliações)
Descrição do aplicativo: o SP156 é um aplicativo utilizado para facilitar a comunicação entre o cidadão e a Prefeitura de São Paulo (PMSP). Os cidadãos podem auxiliar na gestão da cidade a partir da solicitação de serviços, reclamações e denúncias. Além disso, podem acompanhar suas solicitações realizadas em qualquer canal da Solução de Atendimento SP156 (Portal de Atendimento SP156 e Central SP156).

Assim, anteriormente está disposto alguns dos aplicativos existentes de relatos de problemas urbanos dentre os inúmeros que foram utilizados para analisar a forma como as solicitações são realizadas pelo cidadãos. A seção posterior mostra uma taxonomia proposta elaborada com esse estudo.

[2] (GOOGLE, 2019b)
[3] (GOOGLE, 2019d)

3 API Proposta

Esta seção aborda a questão dos problemas urbanos, o estudo realizado para elaboração de uma taxonomia com o intuito de categorizar os problemas das cidades e a API proposta. São expostos os serviços que a API URB oferece e como ela está estruturada, destacando os seus principais requisitos através de um diagrama de classe. Além disso, a modelagem adotada, a orientada a grafos e, por fim, o banco de dados utilizado para o armazenamento dos mesmos.

3.1 Problemas Urbanos

O planeta tem passado por muitas transformações, artificiais e naturais. O crescimento dos centros urbanos tem provido significativos desafios à gestão urbana, com o surgimento de diversos problemas. Segundo AURÉLIO (2019) a palavra problema é definida como aquilo que impede ou dificulta. Assim, ao ser adicionado ao contexto urbano, tem-se os problemas urbanos como causadores da diminuição da qualidade de vida dos cidadãos e dificultadores da obtenção de uma cidade sustentável. Os problemas provêm da saturação das infraestruturas urbanas, dos efeitos ambientais adversos, aspectos socioeconômicos, dentre outros.

Segundo Diniz (2011) os fatores condicionantes para o surgimento dos problemas urbanos são: a falta de administração dos meios físicos ou culturais; a ausência de planejamento; a falta de interesse por parte dos gestores em realizar as ações públicas ágeis; e não levar em consideração o plano diretor. O plano diretor foi instituído pela Constituição Federal de 1988 e os seus princípios estão contidos no Estatuto da Cidade, em que é definido como um instrumento de desenvolvimento e de ordenamento da expansão urbana de um município (REZENDE; ULTRAMARI, 2007).

Os grandes centros urbanos são os mais afetados pelos problemas, detectam-se inúmeros deles, como: os de infraestrutura, ambientais, segurança, lixo, mobilidade e acessibilidade, dentre outros. Embora distintos, os problemas formam uma rede em que um está ligado ao outro. Como exemplo, toma-se a situação em que uma cidade não oferece infraestrutura adequada para ter ou construir ciclovias, logo, essa dificuldade tomará proporções maiores e estará relacionada com a mobilidade e com acessibilidade, dificultando a locomoção de ciclistas.

Assim, diante dos problemas existentes, é necessário que a gestão urbana desempenhe o seu papel, solucionando os problemas, além de realizar um planejamento urbano com finalidade de precaver o surgimento de novas situações problemáticas. De acordo com

HORIZONTE (2015):

> Em 1988, a Constituição Federal instituiu que as cidades e as propriedades precisam cumprir sua função social, ou seja, atender aos interesses da sociedade como um todo, garantindo o bem-estar dos cidadãos. Também foi definido que a gestão das cidades deve ser feita de forma democrática, com a participação da sociedade, nos seus diversos segmentos (HORIZONTE, 2015, p. 15).

Então, conhecer e compreender as dificuldades é essencial para subsidiar a construção de políticas públicas integradas, visando atuarem nas múltiplas causas dos problemas urbanas. O histórico e registro dos problemas, além de sua identificação, são considerados importantes elementos para a qualidade ambiental urbana, assim como para os indicadores de expectativa de vida, infraestrutura urbana, características demográficas e socioeconômicas. Tais auxiliam no desenvolvimento de políticas públicas ligadas a diversos assuntos (MOCHIZUKI; BRESSANE; SALVADOR, 2010; MORATO et al., 2006).

3.2 Taxonomia dos Problemas Urbanos

Diante do estudo realizado em aplicativos que relatam problemas urbanos, além de artigos relacionados, constatou-se a não existência de um padrão para apresentar os problemas, os quais são listados sem nenhum critério. Assim, surgiu a necessidade de separá-los em categorias, a fim de obter um melhor entendimento e organização. Ademais, facilitar a busca de um problema pelo cidadão, resultando na redução do tempo de relato. Com base nisso, produziu-se uma taxonomia, que, a priori, é utilizada para descrever, agrupar e identificar os problemas com as mesmas características e inseridos em um mesmo contexto.

Uma das vantagens de utilizar uma taxonomia é organizar e disponibilizar informações de maneira adequada, a fim de gerar novos conhecimentos. Além disso, propiciar que outras pessoas possam utilizar a mesma expressão para designar um fato, um conceito, ou uma situação a ser registrada, minimizando a ocorrências de conceitos ambíguos.

Para a elaboração da taxonomia proposta efetivou-se um estudo decorrido por meio da análise de aplicativos disponíveis no Google Play, utilizados para realizar o relato de problemas urbanos, dentre eles: SP156[1], Pelas Ruas[2], Problemas urbanos[3], dentre outros. A pesquisa deu ênfase para a maneira como os cidadãos informam os problemas existentes. Além disso, realizou-se a busca de artigos que relatassem os principais problemas enfrentados pelas cidades. A seleção deste estudo teve como base os critérios do trabalho

[1] (GOOGLE, 2019d)
[2] (GOOGLE, 2019b)
[3] (GOOGLE, 2019c)

de BARROS (2015), intitulado como "UTIL: Uma Taxonomia Unificada Para Visualização de Informação". Desse modo, foram definidas as seguintes etapas:

1. **Leitura por Título:** efetivou-se a leitura do título dos trabalhos com intuito de selecionar os artigos relevantes no cenário de gestão urbana e problemas urbanos;
2. **Leitura do Resumo e Palavras-chave:** após a seleção dos artigos pelos títulos, realizou-se uma leitura dos resumos e palavras-chave, objetivando aferir se os trabalhos se adequavam à conjuntura, caso contrário, eram suprimidos dos estudos relevantes;
3. **Leitura Diagonal:** efetivou-se uma leitura diagonal, que consiste na leitura da introdução, seções e conclusões dos artigos para assegurar a exclusão dos trabalhos não relevantes;
4. **Leitura Completa:** a leitura completa dos artigos selecionados para avaliação da importância e para a extração de informações.

Diante da ausência de uma sistematização e categorização dos problemas urbanos, esta pesquisa propõe uma taxonomia, apresentada na Figura 11, estruturada em sete categorias. Cada uma das classes expostas apresenta os seus respectivos problemas urbanos, utilizadas como base para o desenvolvimento de aplicações e de um método fácil para localização das problemáticas.

Para a realização da classificação foram adotados critérios como: analisar as semelhanças e diferenças, presentes entre os problemas urbanos, e estudar os contextos em que estão inseridos. Além disso, um layout, que permite uma visão geral e compacta dos problemas representados por cores distintas para facilitar a interpretação do leitor. A taxonomia divide-se nas seguintes categorias:

- **Infraestrutura:** essa categoria é formada por problemas referentes à estrutura e a serviços básicos que devem ser fornecidos em uma cidade como: vias para deslocamentos de veículos, número de vagas de estacionamento para atender à demanda de veículos, dentre outros. Desse modo, a propiciação de uma boa infraestrutura pelo centro urbano diminui consequentemente as possibilidades de ocorrerem problemas de mobilidade e acessibilidade.
- **Mobilidade e Acessibilidade:** a mobilidade é constituída pelos problemas enfrentados pela população para locomoção nos espaços urbanos. Já a acessibilidade refere-se à dificuldade de acesso que os cidadãos enfrentam para realizar suas atividades cotidianas e deslocamentos.

- **Trânsito:** essa categoria apresenta os principais problemas relacionados ao tráfego público de veículos, congestionamento, sinalização, vias públicas, dentre outros. Como também, abrange as sugestões de possíveis reparos a ser realizados, contribuindo para a harmonização do trânsito e com a qualidade de vida dos cidadãos.

- **Meio Ambiente:** reúne os problemas relacionados à poluição e a má qualidade do ar atmosférico, à emissão de gases poluentes, à excessiva produção de resíduos sólidos, à proliferação de produtos químicos tóxicos, além dos problemas causados por precipitações intensas, que resultam em cheias. Abrange, também, relatos sobre saneamento.

- **Lixo e Limpeza:** esses problemas poderiam estar inclusos na categoria Meio Ambiente, supracitada, porém o lixo causa uma grande quantidade de problemas, assim, requer uma categorização específica. Dessa forma, a classe surge com o intuito de apresentar as dificuldades geradas pelo do lixo, como entulho, ferros velhos, dentre outros. Além disso, a solicitação de limpeza, responsabilidade, em grande parte, da gestão da cidade.

- **Eletricidade:** refere-se aos problemas elétricos das cidades, como postes caídos, reparo de fiação, bem como outros. A iluminação pública possui um papel fundamental na qualidade de vida e segurança de um centro urbano. Algumas ocorrências criminais podem estar atreladas com a falta de iluminação nos perímetros urbanos.

- **Segurança:** abrange os problemas que colocam em risco a vida dos cidadãos, como: manifestações do crime e assaltos; além de áreas de perigo, a exemplo, terrenos propensos à deslizamentos.

Portanto, a taxonomia proposta foi fundamental para o levantamento de requisitos, assim como, a implementação da API, bem como posteriormente será publicada, visando assim a avaliação diante da comunidade científica, com intuito de contribuir no contexto de problemas urbanos das cidades permitindo assim que outros tipos de estudos sejam desenvolvidos utilizando essa taxonomia como referência.

Figura 11 – Taxonomia URB.

Fonte – Autoria própria.

3.3 API URB - Uma API REST para gerenciamento de problemas urbanos

A API proposta neste trabalho foi desenvolvida com o emprego do estilo arquitetural REST. Esse estilo tem sido utilizado para integrar diferentes sistemas de maneira simples e rápida, facilitando o intercâmbio das informações com diferentes linguagens de programação e plataformas de implantação. Assim, é possível realizar a conexão com as aplicações web e mobile, por exemplo, através das especificações e padrões definidos pelas APIs.

A implementação dessa API tem como objetivo fornecer funcionalidades para construir uma base de dados com o intuito de gerenciar os relatos de problemas. Além disso, separar o *back-end* do *front-end*. Para o desenvolvimento de APIs é necessário a definição das funcionalidades que serão providas, assim como, a forma de armazenamento a ser utilizada, isto é, a escolha de um banco de dados. A seguir, a Figura 12 mostra como a API está estruturada dentro de um sistema.

Figura 12 – Representação da API URB em camadas.

Fonte – Autoria própria.

A arquitetura está dividida em três camadas. Na primeira camada, os clientes, que são os requisitantes dos serviços. Na segunda, a API, onde os serviços são providos junto ao *framework* Express e a plataforma Node.js, utilizados para o desenvolvimento das funcionalidades. E, por último, na terceira camada, encontra-se a parte do armazenamento das informações através do banco de dados ArangoDB.

O funcionamento da API acontece da seguinte maneira: uma aplicação envia uma requisição para o servidor, a mensagem deve especificar qual o método será utilizado para o envio. Os métodos HTTP (PUT, GET, ...) indicam qual ação deve ser realizada no

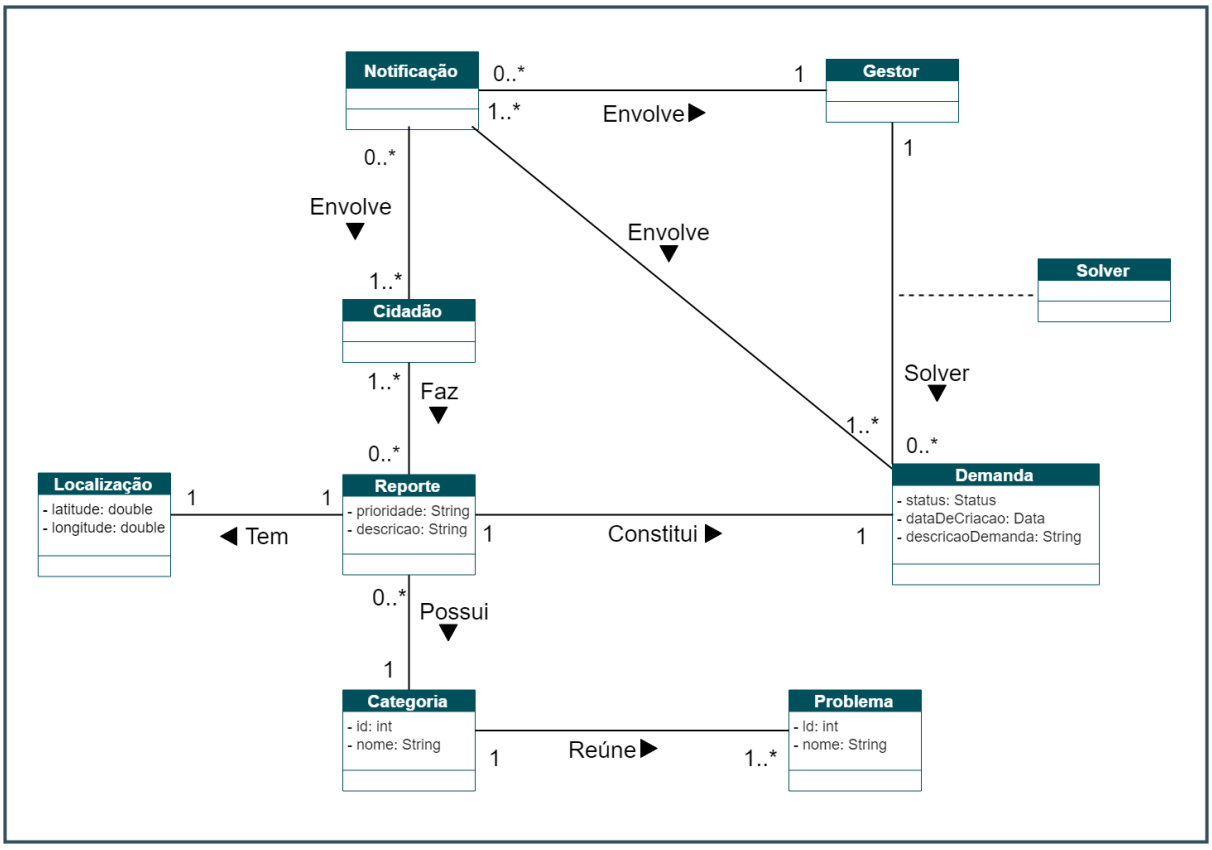
servidor, como, por exemplo, cadastrar um usuário, consultar dados do sistema, realizar atualização dos dados, dentre outras. Após a requisição, obtém-se uma resposta do protocolo HTTP que, através de um código de estado (1xx, 2xx, ...), indica ao cliente sobre o que ocorreu com a requisição realizada no servidor.

A próxima subseção apresenta o diagrama utilizado para realizar o levantamento de requisitos, além dos dados e banco de dados adotados para realizar o armazenamento das informações providas pelos usuários.

3.4 Definição do Modelo de Dados

Esta subseção apresenta o diagrama de classes utilizado no processo de levantamento de requisitos para a implementação da API proposta. A Figura 13 exibe as classes, os seus relacionamentos e os principais atributos da API. Entretanto, novos atributos ou classes podem ser adicionados.

Figura 13 – Diagrama de Classes.



Fonte – Autoria própria.

Reporte é a principal classe da aplicação proposta, pois através dela são geradas as outras classes. O Reporte representa a solicitação de um problema existente. Desse modo,

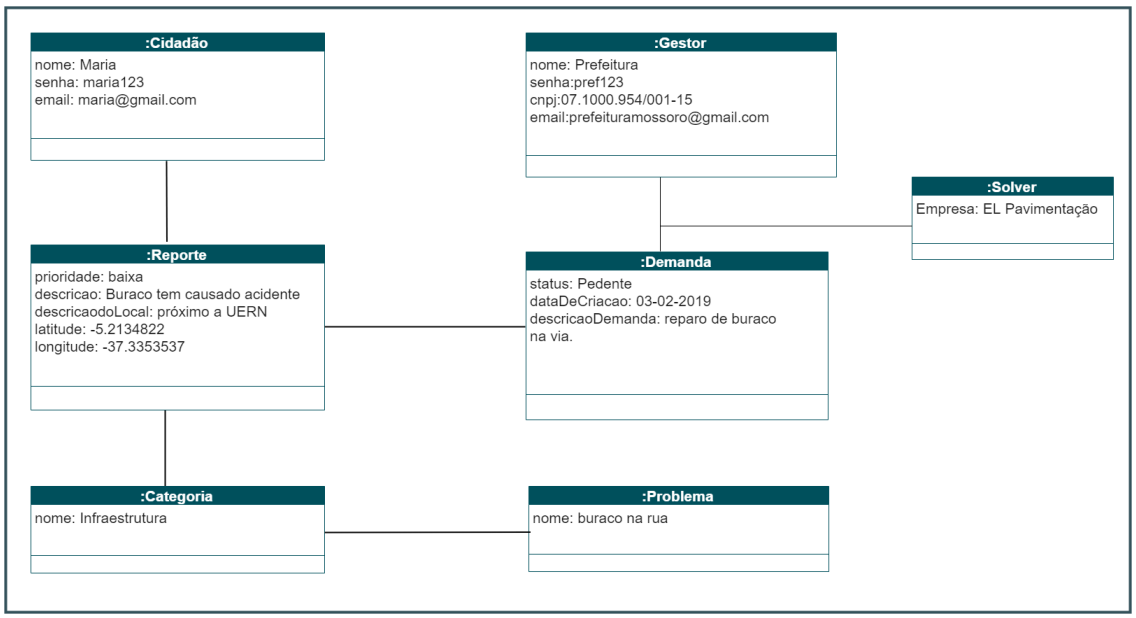
possui como atributos uma descrição referente ao problema, uma prioridade, localização e uma categoria, que reúne os problemas passíveis de ser relatados. A classe Cidadão refere-se aos usuários que irão realizar o Reporte.

A categoria Gestor representa os órgãos públicos, privados e entidades não governamentais responsáveis pela gestão urbana. Já a classe Demanda diz respeito a um Reporte associado a uma data, status e descrição, sendo mantida pelo Gestor e atualizada pelo Solver, que são setores ou empresas responsáveis pela realização dos reparos dos problemas reportados.

A classe Notificação tem como intuito informar e alertar o Cidadão e o Gestor sobre as mudanças resultantes das alterações na descrição e status da Demanda. Essa categoria facilita o acompanhamento e verificação da situação dos relatos encaminhados ao gestor competente pelo cidadão, tornando, assim, possível analisar as decisões tomadas a respeito da situação informado.

Para uma melhor compreensão do modelo demonstrado, a Figura 14 modela através do diagrama de objeto um exemplo de uma cidadã relatando um problema urbano. Propõe-se o seguinte cenário: Maria, uma Cidadã que ao se locomover pela cidade depara-se com um buraco na rua, o Problema. Por ser um problema é passível de categorização, mais especificamente, na Categoria Infraestrutura. Para a realização do relato é necessário o fornecimento da localização por Maria, podendo ser a mesma do problema ou não, e uma descrição o local. Logo após informar esses dados, ela poderá enviar o seu reporte ao Gestor.

Figura 14 – Diagrama de Objeto.



Fonte – Autoria própria.

De acordo com Mossoró (2019) através do seu plano diretor, é de responsabilidade da prefeitura resolver as solicitações de serviço relacionados à infraestrutura urbana. Ao ter as solicitações, o órgão gestor pode gerar uma Demanda que será encaminhada a um Solver, setor ou empresa responsável por realizar os reparos desse tipo de problema, nessa situação, um buraco e, através de uma Notificação, o cidadão é informado se o problema já foi solucionado.

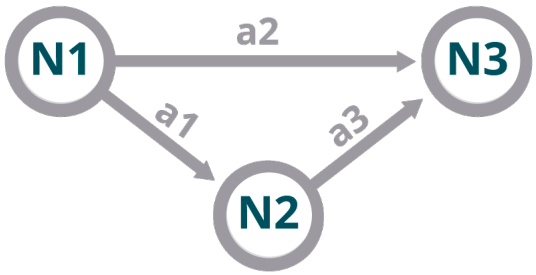
3.5 Banco de Dados

Com base no diagrama de classe apresentado na Figura 13 e Alvarez, Ceci e Gonçalves (2015) pode-se dizer que quando o relacionamento entre as entidades é o ponto principal da modelagem de dados, o modelo orientado a grafos é uma maneira eficaz, além de ser uma estrutura simples para representação das informações. Desse modo, conclui-se que a representação dos dados utilizando esse modelo é uma boa alternativa para compreender os componentes e as relações.

Os bancos de dados orientados a grafos têm como base a teoria dos grafos. Segundo Jaiswal (2013), nesses bancos de dados as informações são apresentadas através de nós e arestas, que apresentam propriedades particulares. Os nós representam os objetos, e as arestas os relacionamentos entre os nós. A Figura 15 mostra um exemplo do modelo em grafos com os seguintes elementos:

- **Nó:** utilizado para representar os objetos ou entidades no banco.
- **Label:** é um tipo de nó, usado para agrupar e possui um nome.
- **Relacionamento:** representa a interconexão entre os nós e também possui nome.

Figura 15 – Representação de um grafo.



Fonte – Autoria própria.

Assim, a API URB dispõe de uma modelagem orientada a grafos, como se pode observar na Figura 16, a organização dos componentes e as suas relações, assim temos os nós que são as coleções: Cidadão, Reporte, Demanda, Gestor, Solver, Problema e Categoria. As arestas estão representadas base um padrão definido por Silva (2017) no seu trabalho de monografia o uso do *has*.

Figura 16 – Modelagem orientada a grafos da API URB.

Fonte – Autoria própria.

A aplicação tem algumas relações representadas através das arestas na Tabela 2. Assim, por meio delas, é possível navegar pelo grafo realizando consultas, como por exemplo, na relação hasCategoriaProblema, dada uma determinada Categoria consegue identificar todos os problemas pertencente a uma Categoria, assim como na relação hasCidadaoReporte obtém as informações qual o Reporte realizado por um Cidadão qualquer.

Tabela 2 – Coleções de Arestas da API URB.

Coleções de Arestas	Descrição
hasDemandaGestor	Demanda possui um Gestor.
hasSolverDemanda	Demanda tem um ou mais Solver.
hasDemandaReporte	Demanda tem um ou mais Reporte.
hasCreateSolver	Gestor cria um contrato com um Solver para atender uma Demanda.
hasReporteProblema	Reporte possui um Problema.
hasCidadaoReporte	Cidadao faz um ou mais Reporte.
hasCategoriaProblema	Categoria tem um ou mais Problema.

Fonte – Autoria própria.

Os bancos de dados orientados a grafos surgiram a partir de tendências como *bigs users*, *big data*, *internet of things* e *cloud computing*, como uma alternativa em relação

aos bancos de dados relacionais (ROBINSON; WEBBER; EIFREM, 2013). Assim, são classificados como um tipo de banco de dados Not Only SQL (NoSQL), aqueles usados para grandes quantidades de informações em diferentes aplicações, como por exemplo: redes sociais, bioinformática, redes de computadores e mecanismos de recomendação.

A primeira vez que o termo NoSQL foi usado ocorreu 1998 para citar um banco de dados relacional open-source que omitia o uso de SQL, o Strozzi NoSQL, criado por Carlo Strozzi (FOWLER, 2013). Assim, esse termo não define com precisão esses bancos, mas reúne aqueles com características em comum, como: não-relacional, distribuído, de código aberto, escalável horizontalmente, esquemas flexíveis, suporte à replicação nativo e acesso via APIs simples (DIANA; GEROSA, 2010).

Conforme Robinson, Webber e Eifrem (2013) a utilização de banco de dados orientados a grafos possui algumas vantagens tais como: facilitar as modelagens de dados sem que haja necessidade de estar estruturando o esquema de dados e apresentar um alto desempenho independente do total de conjuntos de dados. O banco de dados orientado a grafo utilizado para armazenar os dados da API proposta é o ArangoDB.

3.6 ArangoDB

A primeira versão do ArangoDB foi lançada em 2011 pela ArangoDB GmbH. A implementação do mesmo foi desenvolvida em C++. O ArangoDB é um banco de dados NoSQL, sendo híbrido, o qual dá suporte a alguns modelos como: chave-valor, orientado a documentos e grafos. Além disso, é código aberto (ARANGODB, 2019).

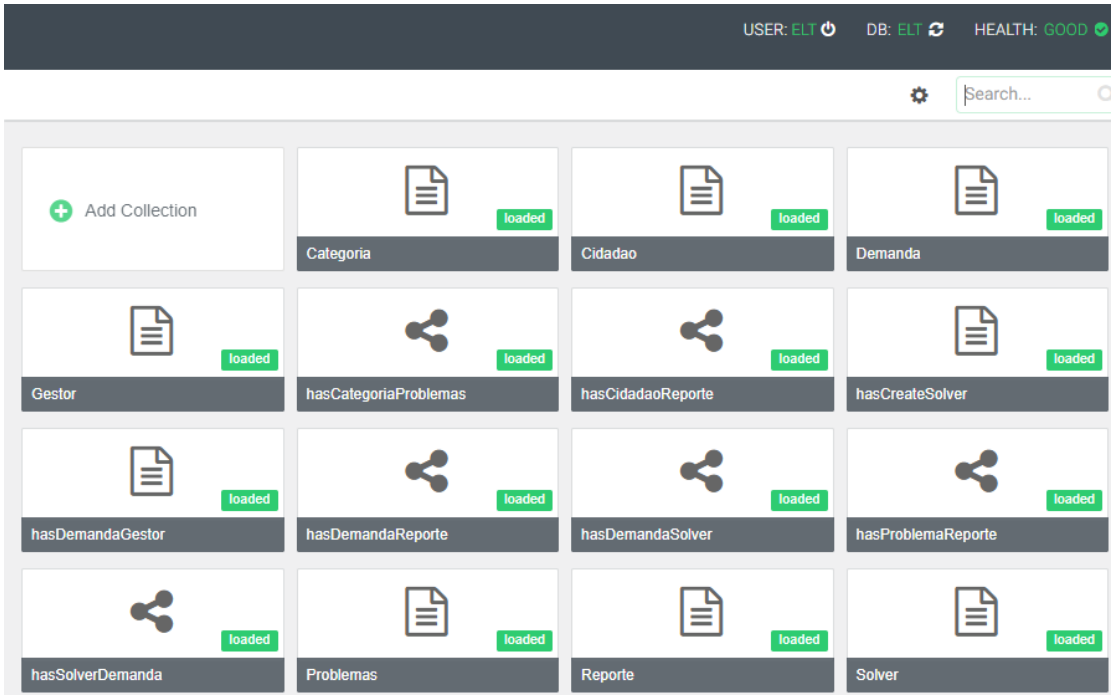
Uma das vantagens é a simplicidade de configuração e a instalação desse banco, pois se expande para diferentes sistemas operacionais. O funcionamento ocorre através da realização de uma simples instalação do software para executar o banco de dados. Além de possuir uma interface simples, seu grupo de desenvolvimento é bastante ativo, sempre com melhorias e atualizações, dispondo de uma documentação com fácil aprendizado.

Por meio desse banco de dados é possível realizar a combinação de diferentes modelos numa consulta. O formato dos dados é representado em JSON, através do Transmission Control (TCP), utilizando protocolo HTTP. Além dessas características, Sousa (2016), enumera outras, tais como:

- Schema-Free: consiste na combinação da eficiência de espaço do MySQL com a performance do NoSQL;
- Servidor de Aplicativos: integração do banco de dados entre a aplicação com uso do máximo de rendimento;
- JavaScript: uso de uma única linguagem do navegador até o back-end;

- Escolha livre de índice: permite a seleção do tipo de index;
- Transações: possibilitam a execução de consultas em múltiplos coleções ou documentos com opção de consistência transacional e isolamento;
- Replicação: pode ser utilizada em uma configuração mestre-escravo;
- Código Aberto.

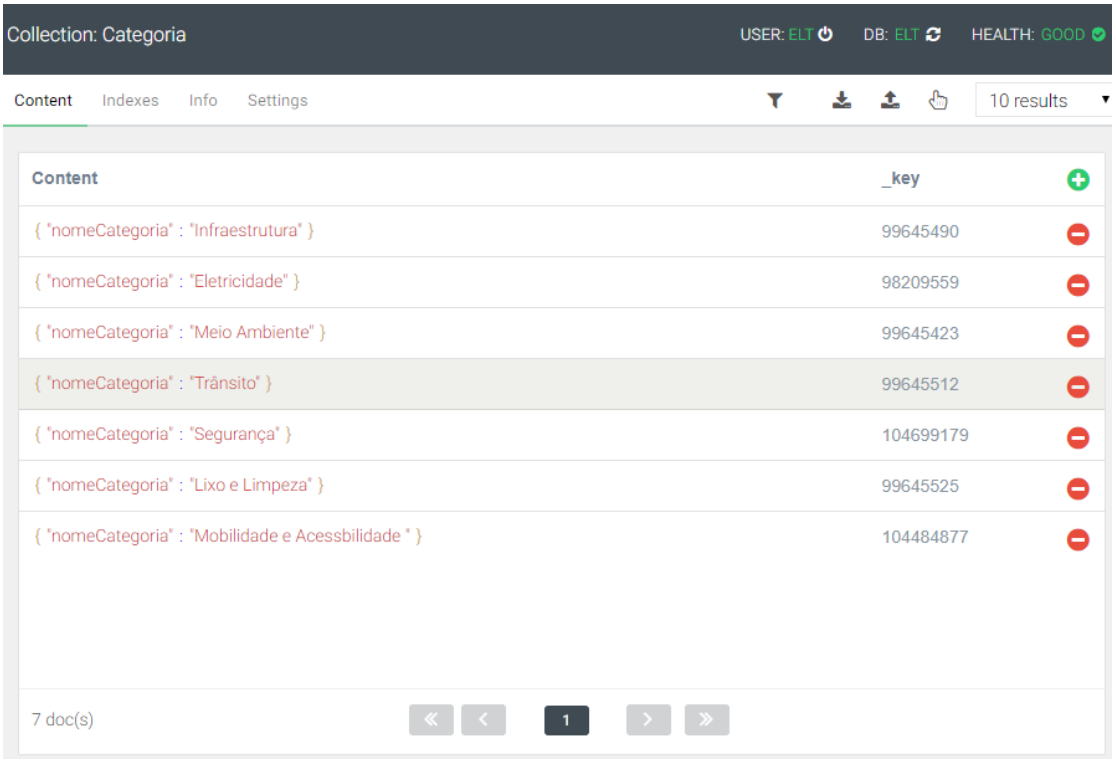
Figura 17 – As coleções da API URB no ArangoDB.



Fonte – Autoria própria.

Esse banco é composto por documentos agrupados em coleções. Uma coleção pode conter um ou mais documentos. Para uma melhor compreensão, pode-se comparar com os bancos relacionais, os documentos com as tuplas (linhas) e as coleções que se assemelham às tabelas. A principal diferença consiste no ArangoDB permitir a criação de cada documento com seu próprio esquema, uma vez que os relacionais apresentam colunas como definição para um esquema de dados, devendo ser empregado por todas as tuplas (SILVA, 2017). A Figura 17 mostra a interface do banco de dados Arango DB apresentando as 7 coleções. Ele é representado com ícone de documento, que são os nós e as relações, no caso, as arestas, somando o total de 6 expostas como grafos. Já a Figura 18 mostra os documentos da coleção Categoria.

Figura 18 – Os documentos da coleção Categoria.



Fonte – Autoria própria.

Ainda conforme Silva (2017), no ArangoDB existem dois tipos de coleções: as coleções de documentos que representam os vértices no cenário de grafos, e as coleções de arestas, também documentos, porém com dois atributos o _from (origem) e _to (destino), utilizados para gerar as relações entre os documentos. Para realizar todas as manipulações esse banco de dados utiliza a linguagem AQL.

A AQL é fundamentada na linguagem tradicional SQL, mas possui recursos adicionais por ser dinâmica e ter o armazenamento de documentos, construídos através do uso do SELECT. Os documentos passíveis de agrupamento também podem ser consultados através aplicação do ArangoDB. As consultas são transmitidas via HTTP e o formato de resposta é o JSON.

Logo, o ArangoDB entra no seguinte contexto para armazenar todas as informações do Reporte e, assim, ser possível criar históricos. A API se comunica com ArangoDB através dos módulos disponibilizados via Node Package Manager (NPM), um gerenciador de pacotes para linguagem de programação JavaScript. Esses módulos fornecem a comunicação entre a API e o banco de dados. A configuração que permite a conexão com o banco de dados é realizada através do *framework* Express utilizando o seguinte código:

Código 3 – Conexão da API com o Banco de Dados.

```
module.exports = function () {
  var arangojs = require("arangojs");

  var username = "elt";
  var password = "senha do banco";
  var host = "lordi.uern.br";
  var port = "8529";
    var database = "elt";

  var db = new arangojs({
    url: `http://${username}:${password}@${host}:${port}`,
    databaseName: database
  });
  return db;
}
```

Fonte – Autoria própria.

Portanto, para a execução do banco de dados, utiliza-se de uma porta que foi disponibilizada pelo Laboratório de Redes e Sistemas Distribuídos (LORDI), da Universidade do Estado do Rio Grande do Norte (UERN), para instalação do ArangoDB. Com o banco de dados online e em funcionamento, tornou-se possível armazenar e gerenciar as informações nessa instância utilizando a API proposta.

4 Implementação e Validação da API URB

Nesta seção, serão exibidos os aspectos relevantes utilizados para a implementação da API URB: uma API REST para gerenciamento de problemas urbanos, assim como as tecnologias e funcionalidades com os princípios REST. Por fim, a validação da API proposta através de aplicação web e *mobile*.

Para o desenvolvimento da API, fez-se necessário realizar um estudo sobre a forma como os cidadãos relatam os problemas urbanos aos órgãos responsáveis, além de analisar alguns aplicativos, resultando em uma taxonomia, fundamentais para o desenvolvimento dessa API. Além disso, emergiu-se a necessidade de um estudo das tecnologias para realizar a implementação.

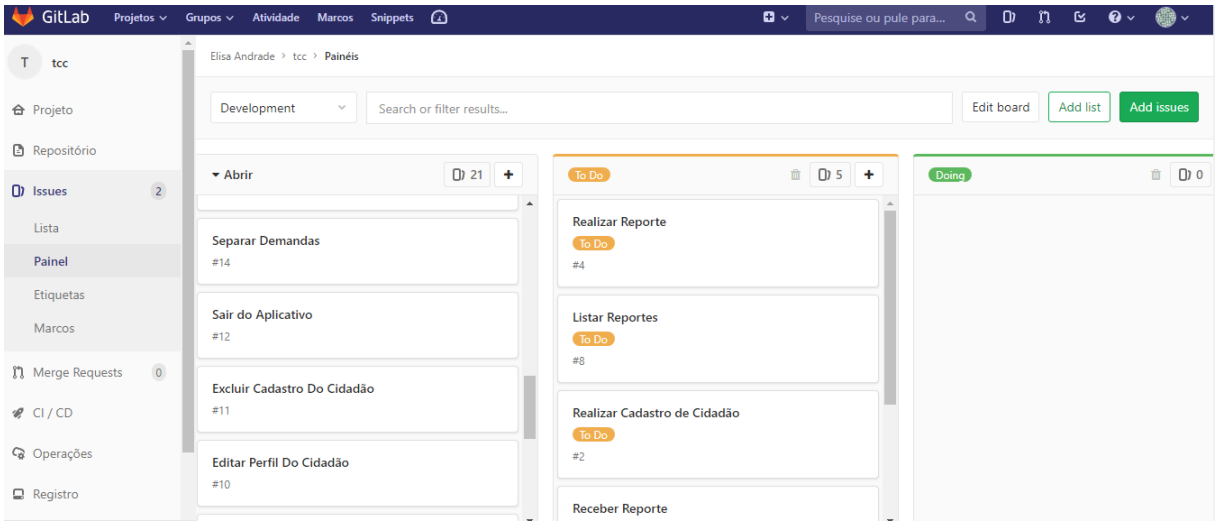
Após a definição dos requisitos funcionais e a modelagem dos dados orientada a grafos, foram definidas as tecnologias que seriam utilizadas para a implementação da API URB, sendo elas a linguagem JavaScript na plataforma Node.js e a *framework* Express como auxílio.

O Node.js possibilitou o desenvolvimento da API URB e a execução de maneira rápida, assim como o processamento de uma grande quantidade de dados, quando comparados com APIs em outras linguagens de programação. Além disso, o servidor HTTP do Node.js, com o uso do Express, realizou de modo simples as configurações necessárias para a comunicação com banco de dados, simplificando o uso da API. Para autenticação utilizou-se O Json Web Token (JWT). Já o como recurso para armazenamento das informações utilizou-se o ArangoDB, conforme descrito na seção 3 deste trabalho.

Na fase de desenvolvimento da API URB, proposta por este estudo, utilizou-se como editor do código fonte o *Visual Code Studio*, além da ferramenta GitLab, utilizada para realizar o acompanhamento das funcionalidades implementadas de maneira dinâmica e intuitiva. Por meio do serviço de etiqueta provido pelo GitLab, realizou-se as listagens do que precisava ser implementado, ou seja, todo o controle do desenvolvimento do projeto. Com essa ferramenta, efetuou-se o versionamento da API, permitindo criar históricos e facilitando realizar possíveis alterações no código quando for preciso, como mostra a Figura 19.

Assim, por meio dessa ferramenta, viabilizou-se a atribuição de prioridades para implementação das funcionalidades e, até mesmo, a inclusão conforme a necessidade. Dessa maneira, tornou-se possível o gerenciamento das tarefas, além da organização do tempo necessário para cumprir as metas, reduzindo os riscos/custos e aumentando a produtividade. Ademais, utilizou-se de um diagrama de componentes para uma melhor representação.

Figura 19 – Uso da Ferramenta GitLab.

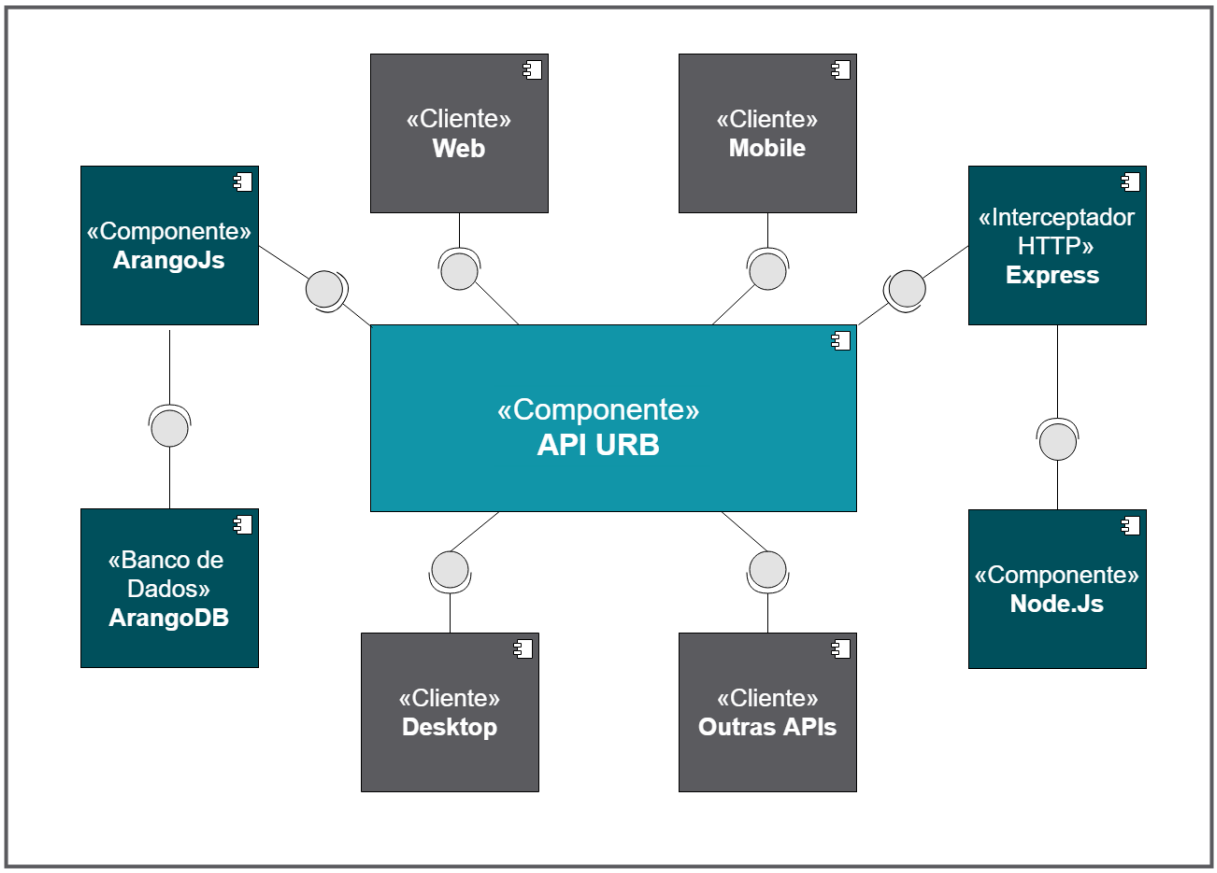


Fonte – Autoria própria.

O diagrama de componentes, mostrado na Figura 20, tem como finalidade evidenciar como os componentes da API URB estão estruturados, bem como a interação entre os mesmos para que a API funcione de maneira adequada. O componente principal, posicionado no centro e representado em cor azul claro, simboliza a API proposta neste trabalho. Já os recursos em azul escuro são aqueles utilizados para a implementação, como o módulo Arango.js, usado para fazer a conexão do banco de dados com a API; o *Framework* Express, que auxilia o Node.js, plataforma utilizada para implementar as funcionalidades; e, por fim, os componentes em cinza, referem-se às diversas aplicações, como web, mobile, desktop ou outras APIs, que podem requisitar os serviços providos.

A API URB propõe como diferencial para as aplicações uma maneira diferente ao realizar o relato de problemas urbanos. Tendo em vista que um dos conceitos principais dessa aplicação é o Reporte, que contém coordenadas geográficas referentes a latitude e longitude do usuário, que serão capturadas pelos clientes consumidores da API. Assim, por meio das informações fornecidas pelo cliente acerca da sua localização, a API realiza uma consulta no banco de dados e lista os Reportes já efetuados por meio das Geo *Functions*, consultas que a linguagem ArangoDB Query Language (AQL) fornece para filtrar os dados com base em índices geográficos. Essas buscas permitem exibir os Reportes, localizados dentro de um raio de até dois quilômetros - por exemplo, possibilitando que os usuários vejam os outros relatos, assim como analisar se de fato as informações estão corretas.

Figura 20 – Diagrama de componentes da API URB.



Fonte – Autoria própria.

4.1 Tecnologias Utilizadas

A seguir, será apresentado uma breve descrição sobre as tecnologias utilizadas para implementação da API URB, com o objetivo de compreender a utilidade de cada tecnologia para o funcionamento da API. Esses recursos foram escolhidos por apresentarem um bom desempenho, bem como um ótimo suporte através de comunidades e de suas documentações.

4.1.1 JavaScript

A linguagem de programação JavaScript surgiu em 1995 para ser executada no navegador Netscape e, posteriormente, foi adaptada para a maioria dos navegadores Web (HAVERBEKE, 2014). No início, idealizado para os navegadores atualizando o conteúdo dinamicamente. No entanto, o surgimento de novas tecnologias, como o Node.js e NPM, contribuiu bastante para que essa linguagem fosse ainda mais utilizada e com finalidades além das previstas inicialmente.

A *Stack OverFlow* a cada ano realiza pesquisas com desenvolvedores sobre suas tecnologias favoritas, até suas preferências de trabalho. No ano de 2018, acerca da pesquisa realizada, afirmou-se que "Mais de 100.000 desenvolvedores fizeram a pesquisa", destacando o JavaScript como uma das mais populares entre as linguagens de programação utilizadas pelos desenvolvedores. Além disso, ressaltou a ocorrência do fato, sendo o seu sexto ano consecutivo, em 2018 com 69,8% dos entrevistados demonstrando sua preferência pela linguagem (OVERFLOW, 2019).

4.1.2 Node.js

O Node.js foi desenvolvido por Ryan Dahl no ano de 2009, trata-se de uma plataforma de desenvolvimento de aplicações Web que trabalha sobre o motor JavaScript do V8, um interpretador JavaScript, desenvolvido pelo Google e utilizado no navegador Google Chrome. Como sucedeu em C++ possui código aberto. Assim, através do V8, é possível acelerar o desempenho de uma aplicação ao compilar o código em JavaScript diretamente para o código binário antes de executar, sendo tão rápido quanto um código escrito em uma linguagem de baixo nível (SEVERANCE, 2012).

O Node.js trabalha com uma abordagem não obstrutiva, com *single-thread* orientado a eventos. Além disso, é bastante utilizado pelo fato de permitir ao desenvolvedor trabalhar com JavaScript no *front-end* e no *back-end*. Ele é executado por um único processo, o que se torna uma das suas grandes vantagens em relação às outras linguagens.

A abordagem orientada a eventos é utilizada para o desenvolvimento de aplicações com o Node.js. A programação orientada a eventos se difere dos demais paradigmas de programação, pois segue um fluxo padronizado. O fluxo desse sistema é indicado por meio de disparos ou indicações externas, chamados de eventos. Portanto, o desenvolvedor precisa ter o conhecimento dos eventos disparados, para que, assim, consiga saber como tratar esse evento e realizar as operações necessárias (SEVERANCE, 2012).

O Node.js oferece um alto desempenho e alta escalabilidade para as aplicações. Dessa maneira, ao desenvolver com o Node.js e o estilo arquitetural REST, emerge a junção da agilidade do REST com a escalabilidade e velocidade do Node.js, tornando possível desenvolver Web Services com um bom desempenho, além de fornecer serviços a uma grande quantidade de usuários, diferentemente do que é disponibilizado pelas arquiteturas convencionais.

4.1.3 Express

O Express é um *framework* desenvolvido em JavaScript para o Node.js, disposto por meio do NPM, sendo minimalista e flexível, além de fornecer um conjunto de recursos fundamentais para aplicativos web e *mobile* (EXPRESS, 2019). Esse serviço torna possível

o desenvolvimento de APIs que recebem requisições HTTP de forma simples, facilita a conexão com o banco de dados, como utilizado nessa implementação exposta no Código 3. Além de oferecer maneiras de reutilizar código elegantemente, dispõe de uma estrutura semelhante ao modelo *Model-view-controller* (MVC) para desenvolvimento das aplicações da web (MARDAN, 2014, p.3). Segundo Pereira (2016) as principais vantagens do Express são:

- Routing robusto;
- Facilmente integrável com os principais Template
- Engines;
- Código minimalista;
- Trabalha com conceito de minimalista;
- Possui uma grande lista de middlewares 3rd-party;
- Content Negotiation;
- Adota padrões e boas práticas de serviços REST.

Assim, o que foi implementado com o uso do Express pode ser realizado apenas com o Node.js, porém de uma forma mais trabalhosa. Desse modo, a função que o Express desempenha para o Node.js é a mesma que o Laravel proporciona para o PHP, o Ruby on Rails fornece ao Ruby e Django provê para o Python.

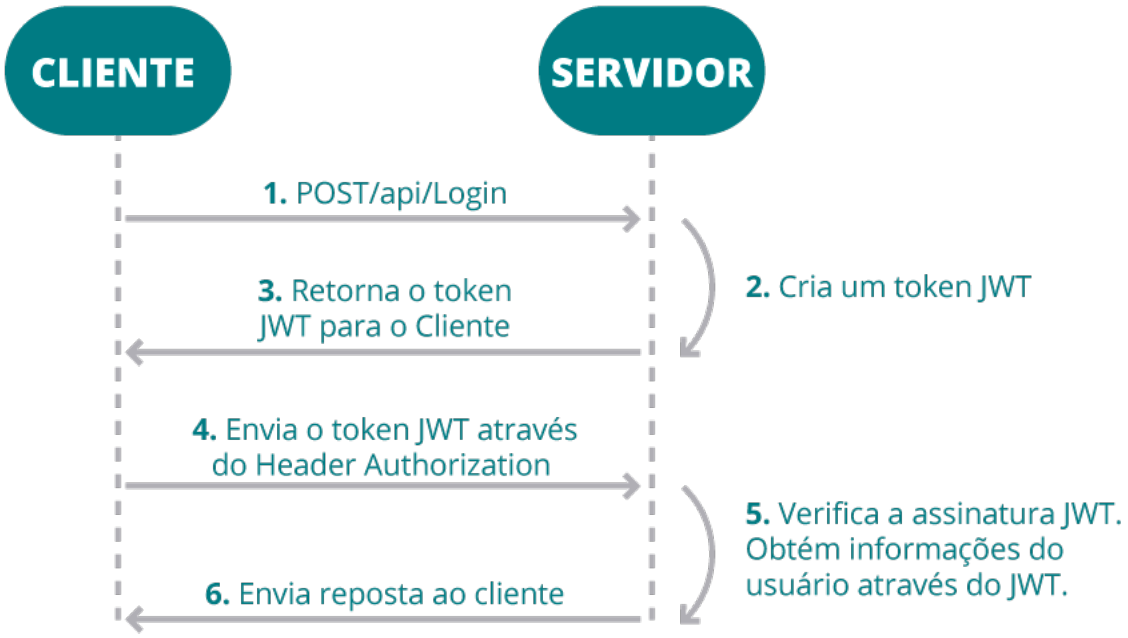
4.1.4 JWT

O JSON Web Tokens, baseado no padrão RFC7519 da W3C, é uma estratégia segura e simples para implementar a autenticação para APIs RESTful, baseado no tráfego em formato JSON, entre o cliente e servidor.

Segundo Oliveira (2018), um *token* JWT é formado por três partes: a primeira é o *header*, onde estão as informações acerca do algoritmo utilizado para criptografar o *token*; a segunda parte corresponde ao *payload*, local de armazenamento das informações de usuário, as quais são utilizadas para gerar um *hash* único; a última parte trata-se do *signature*, que é responsável por realizar o armazenamento da chave criptografada com base no algoritmo armazenado no *header*.

Para um melhor entendimento, a autenticação implementada nessa API e seu funcionamento são mostrados na Figura 21, destacado no fluxo da autenticação e, posteriormente, elencando o que significa cada etapa.

Figura 21 – Fluxo de Autenticação do JWT.



Fonte – Adaptado de CRUZ (2019).

1. Um aplicação web ou mobile, cliente no caso, realiza uma requisição através do método POST uma única vez, enviando as suas informações o e-mail e senha.

2. A API URB valida as credenciais, caso estejam todas corretas retornam ao cliente um JSON com o *token* criado, o qual codifica as informações de um Cidadão ou Gestor logados na aplicação.

3. Ao receber esse *token*, o cliente pode armazená-lo da forma que quiser, mas que seja por meio de *LocalStorage*, *Cookies* ou por outros mecanismos de armazenamento *client-side*;

4. Quando o cliente acessar uma rota é necessário que de fato esteja autenticado, assim, ao enviar o *token* para a API URB irá autenticar e liberar o consumo de dados;

5. A API URB valida um *token* para permitir uma requisição do cliente;

6. Será enviada uma resposta ao cliente informado através do código HTTP.

4.2 Validação

Esta subseção evidencia os testes realizados com os métodos da API e, consequentemente, a validação dos mesmos. O principal objetivo é testar o comportamento dos métodos da API, por meio da validação com a inserção de valores, e, assim, ao ser validado, obter dados e uma resposta como o código *status* retornado.

Testar uma requisição consiste em verificar se a estrutura de dados enviada está conforme o que foi estabelecido, e se o retorno está conforme ao que foi solicitado, respondendo à requisição com o código apropriado. Para realizar os testes da API proposta neste trabalho utilizou-se a ferramenta Postman.

4.2.1 Postman

O Postman, criado em 2012, é muito utilizado pelos desenvolvedores de APIs para realizar pedidos através de métodos HTTP, o qual oferece uma interface simples e intuitiva, que facilita os testes e a depuração dos serviços REST, sendo possível realizar solicitações HTTP como GET, POST, PUT e DELETE. Além disso, pode-se realizar o teste de performance de uma API executados um conjunto de solicitações para verificar seu desempenho de resposta. As figuras apresentadas posteriormente utilizam essa ferramenta.

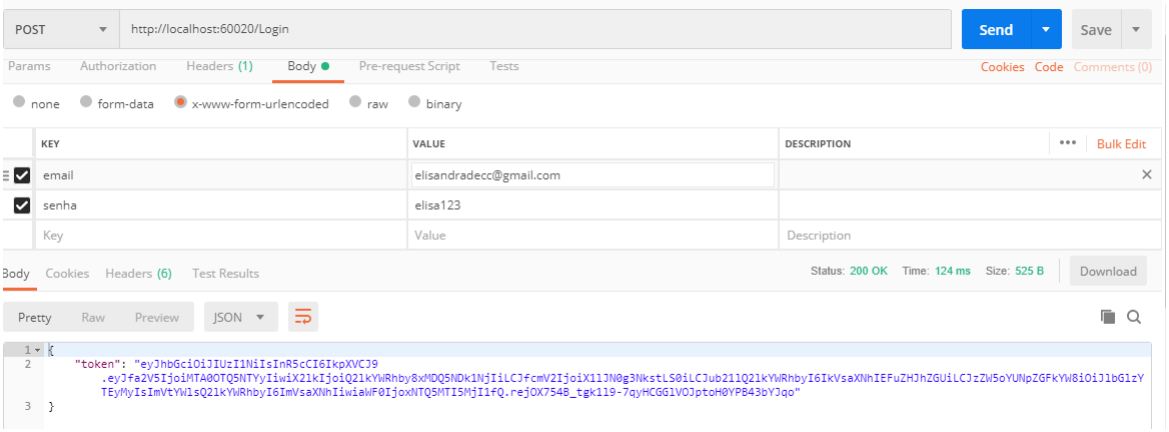
4.2.2 Rotas

As rotas são os pontos de acesso disponibilizados de uma aplicação para que o cliente utilize para consumir os serviços providos por uma API REST, sendo classificadas em recursos como os principais tópicos disponibilizados. Assim, segundo Fielding e Taylor (2000) quando os recursos são disponibilizados por (URI) fazem parte de uma restrição da interface uniforme propostas pelo mesmo, os quais servem como ponto de entrada para o consumo da API.

Para mostrar a implementação e funcionamento da API URB, será exposto os serviços implementados, bem como o funcionamento das principais rotas. O detalhamento das mesmas encontra-se no Apêndice A. Os principais serviços que constituem a aplicação API Urb são:

- **Realizar Login** Realizado através da implementação do método HTTP POST, cuja a resposta retorna um objeto JSON. Permite ao usuário fazer o login ao informar o email e senha como demostrado na Figura 22, mas só é permitido realizar o login se o usuário estiver previamente cadastrado, assim, para que ocorra a interação com o servidor, e para a utilização deste serviço o cliente deve fornecer as informações conforme solicitadas, logo, a API gera um *token* e acontece o fluxo de autenticação citado anteriormente.

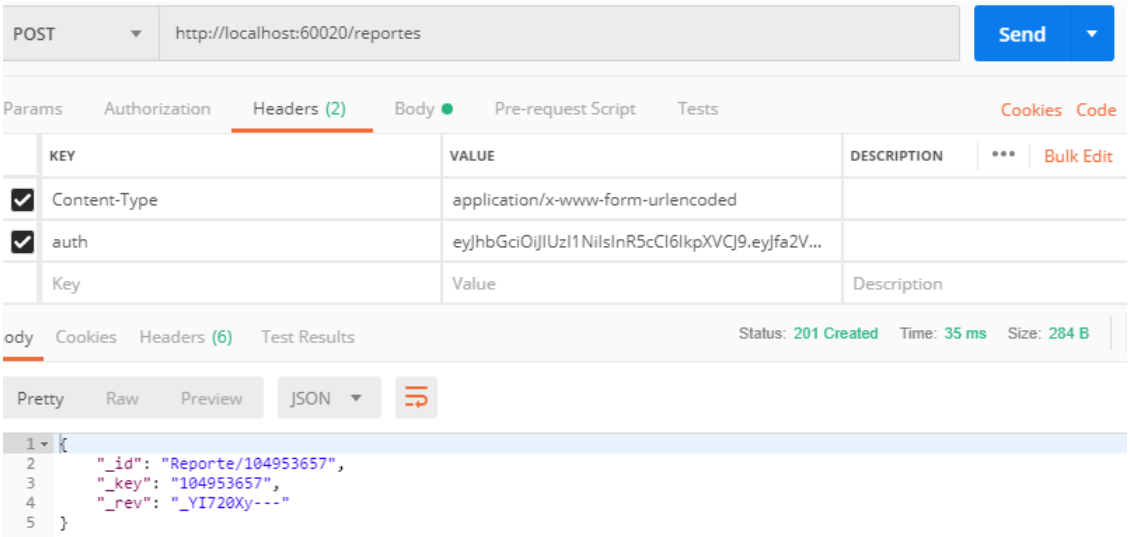
Figura 22 – Realizar Login com Token.



Fonte – Autoria própria.

Com o uso da autenticação e após ser gerado o *token*, o cliente envia no campo *Auth* do cabeçalho do protocolo HTTP o *token* que foi armazenado, que permite a API identificar quem é o usuário solicitante, além do mais constata se tem permissão ou não para acessar a informação desejada, como por exemplo o usuário só poderá realizar um Reporte se tiver efetuado o login na aplicação como apresentado na Figura 23.

Figura 23 – Requisição com o Token.

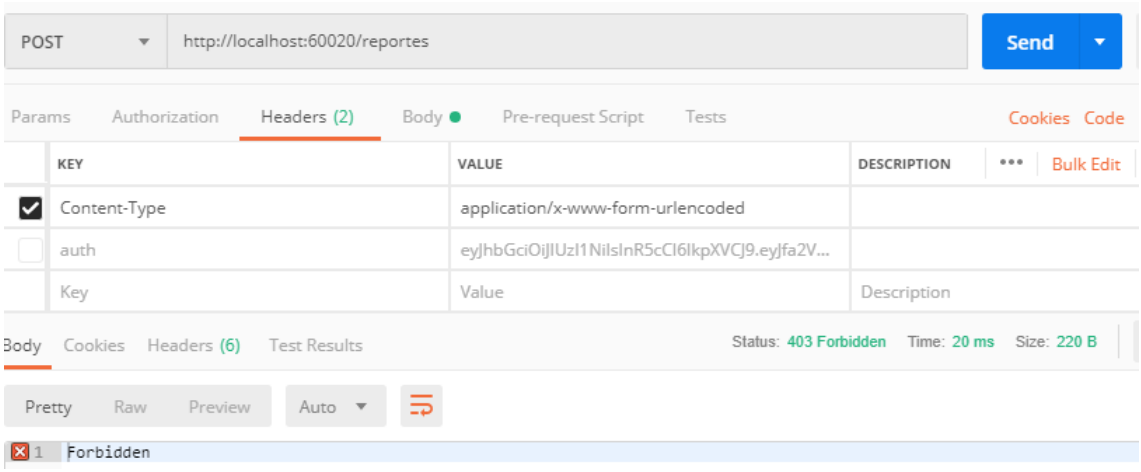


Fonte – Autoria própria.

Com o uso do JWT para autenticação, através do *token*, impede que usuários não cadastrados sejam bloqueados de acessar os serviços da API. Na Figura 23, um

usuário deseja realizar um Reporte, mas não está autenticado, ou seja, não informa o *token*, assim não tem acesso e a resposta para este caso vai ser o código 403, ou seja, *Forbidden*, que significa proibido.

Figura 24 – Requisição não efetuada.



Fonte – Autoria própria.

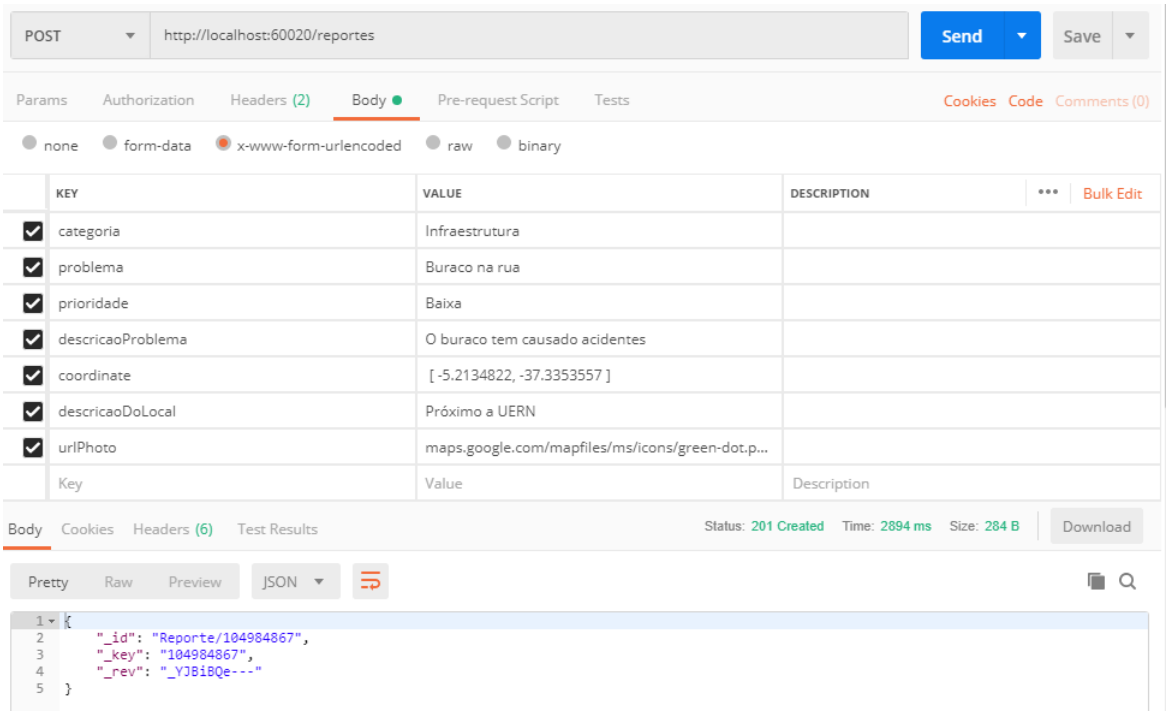
- **Criar um Reporte** Refere-se a um relatório que contém as informações sobre um relato para ser realizado por um cidadão, assim para criar um Reporte através da API, realiza-se uma requisição usando o método POST no recurso reportes. Como mostrado na Figura 25.

  No entanto, o cliente deve informar no corpo da mensagem um objeto JSON, com os atributos: indicar a categoria do problema, o problema que deseja informar, a sua prioridade como por exemplo se é alta, média ou baixa, fica ao seu critério descrever sobre o problema, deve fornecer sua localização, e caso não esteja no local, pode escrever sobre local que o problema encontra-se e arquivar uma imagem.

- **Criar uma Categoria** A categoria é uma forma de organizar os problemas, assim o cliente pode criar suas categorias como achar melhor categorizar, no caso desta API convencionou-se setes categorias citadas anteriormente, para criar uma categoria usa-se o método POST e recebe como atributo o nome da categoria como exibe a Figura 26.

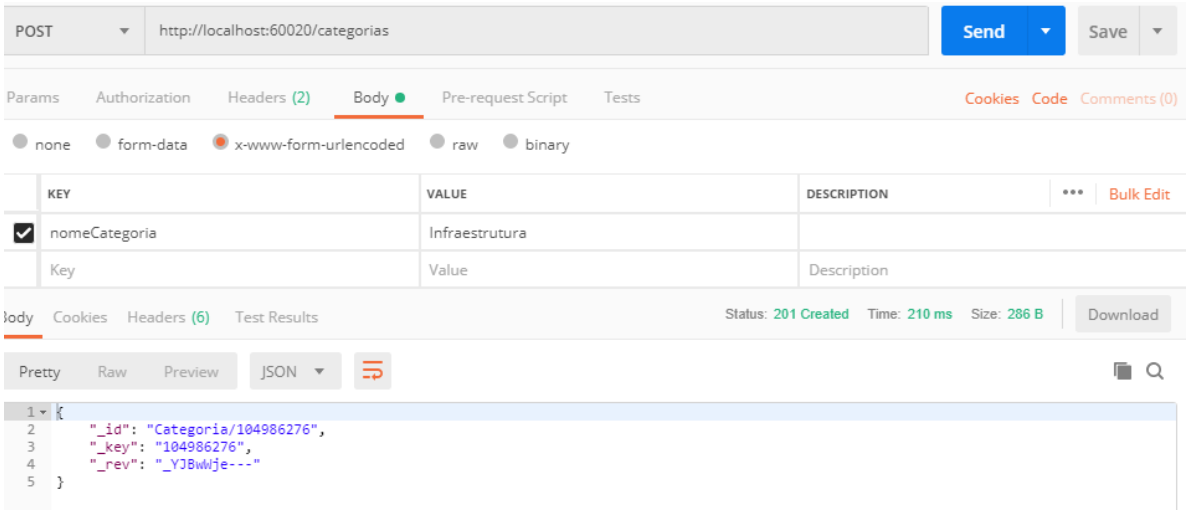
- **Criar um Problema** O problema corresponde aos problemas urbanos sendo a informação principal do Reporte, assim cadastra-se um problema com o método POST, o cliente faz uma requisição e informando como atributo o nome do problema, como é visto na Figura 27.

Figura 25 – Reporte criado com sucesso.



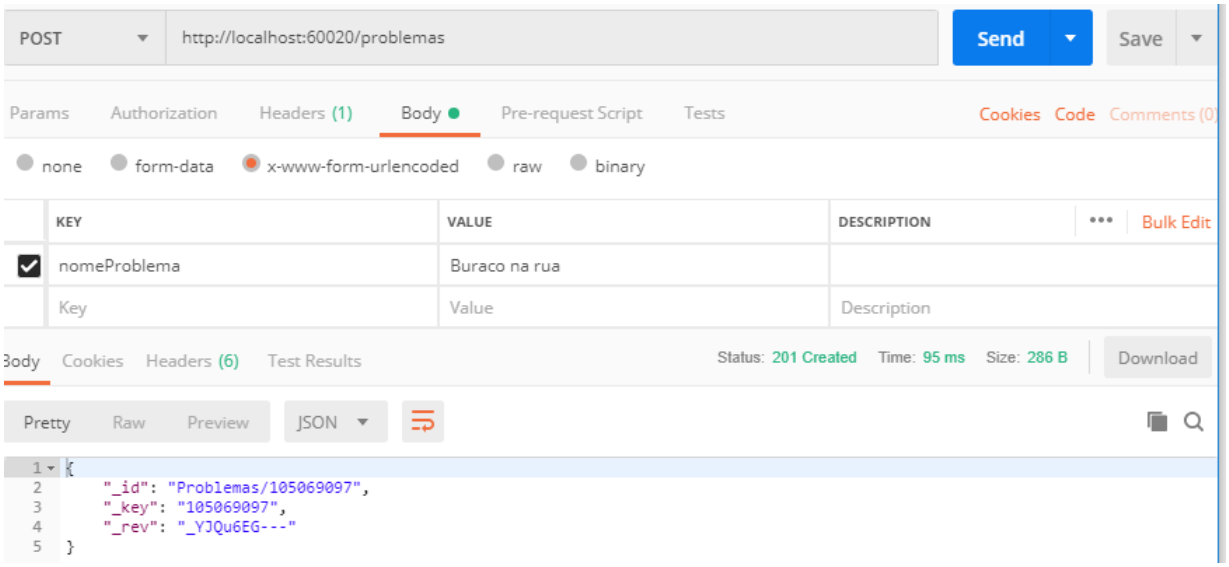
Fonte – Autoria própria.

Figura 26 – Categoria criada com sucesso.



Fonte – Autoria própria.

Figura 27 – Problema criado com sucesso.



Fonte – Autoria própria.

- **Criar uma Demanda** A demanda é criada a partir das informações de um Reporte, essas informações são encaminhadas para um Gestor que envia para um Solver, ou seja para um solucionador que pode ser um setor ou uma empresa que fará o reparo do problema relato, assim para criar uma demanda usa-se o método POST e ao realizar uma requisição deve ser informados os seguintes atributos uma descrição sobre demanda, a categoria, o problema, a prioridade, coordinate e a descrição do local.

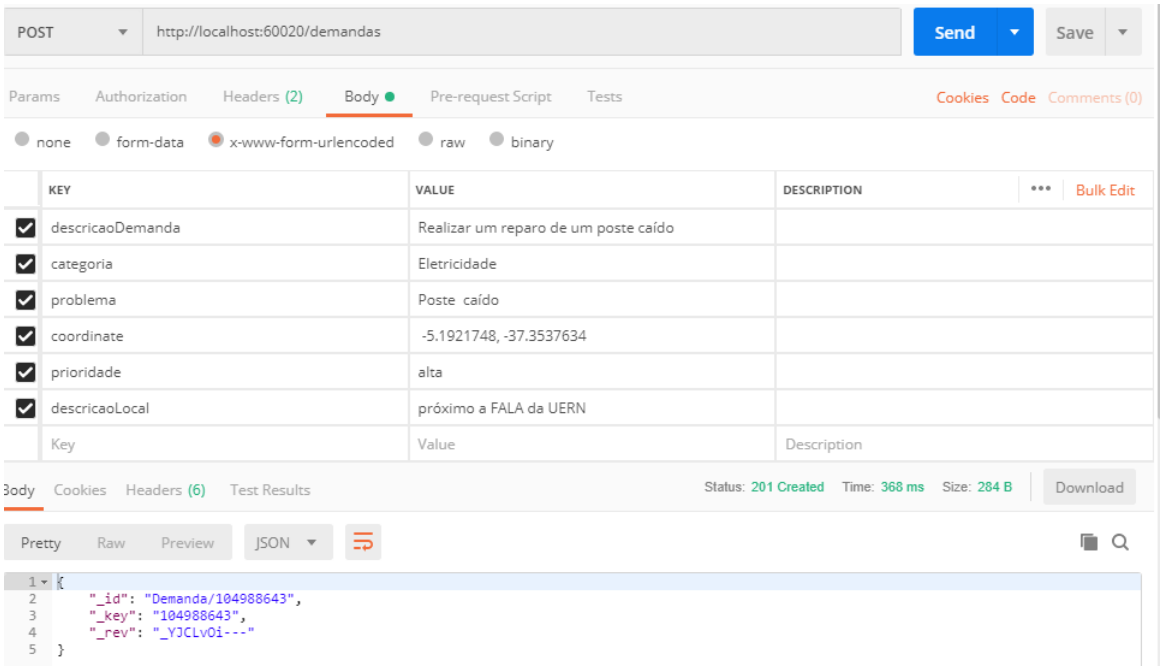
  Assim, para todas as requisições realizadas que foram mostradas anteriormente é exibido o código status 200 OK indicando que a requisição foi bem sucedida, e recebendo um objeto que apresenta um JSON com três informações referentes aos documentos, que são o identificador do documento: _id, a chave-primária: _key, e a revisão do documento: _rev, os quais são gerados a cada transação efetuada no banco de dados.

  As rotas supracitadas referem-se às coleções para mostrar a validação das relações da API será utilizada duas arestas que foram implementadas, com base na modelagem proposta, a orientada a grafos.

- **hasCategoriaProblema** Através dessa relação, torna possível exibir os problemas que estão relacionados a uma categoria, bem como listar esses problemas. O cliente deve realizar uma requisição utilizando o método GET e passar como parâmetro origem com o id da categoria conforme mostrado na Figura 29.

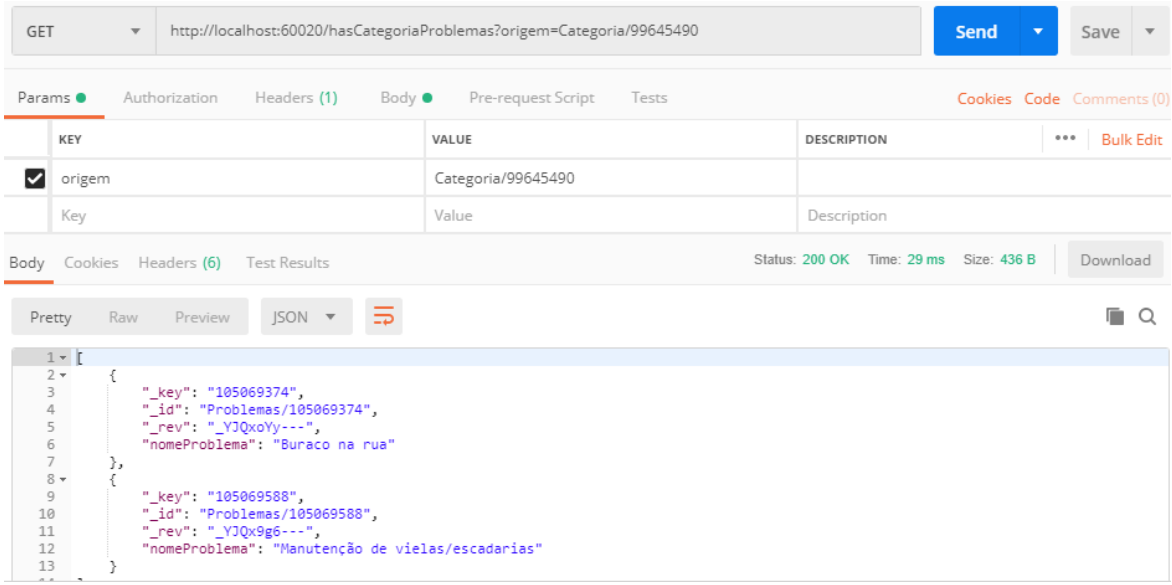
  Com essa aresta, permite exibir os reportes que um determinado cidadão realizou,e

Figura 28 – Demanda criada com sucesso.



Fonte – Autoria própria.

Figura 29 – Listar problemas através de uma categoria.



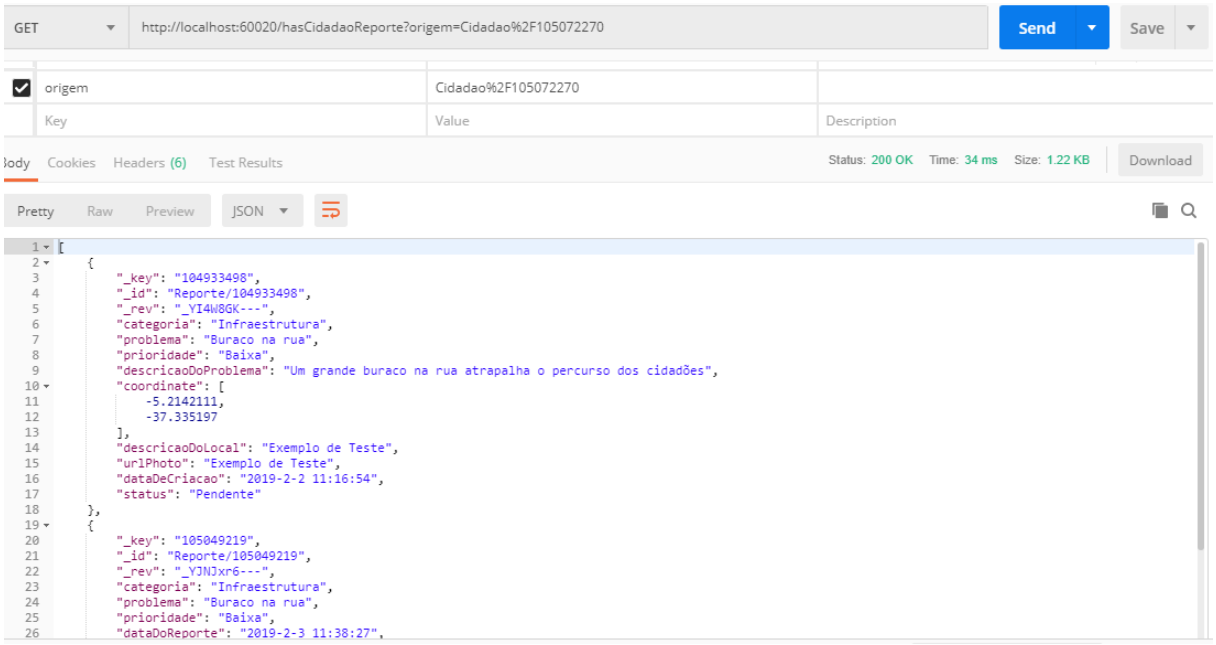
Fonte – Autoria própria.

listá-los, para isso o cliente ao fazer uma requisição deve usar o método GET e passar como parâmetro origem com o id do Cidadão de acordo com a Figura 30.

- **hasCidadaoReporte**

  Com essa aresta, permite exibir os reportes que um determinado cidadão realizou,e listá-los, para isso o cliente ao fazer uma requisição deve usar o método GET e passar como parâmetro origem com o id do Cidadão de acordo com a Figura 30.

Figura 30 – Listar Reportes realizado por um Cidadão.



Fonte – Autoria própria.

Para a realização dos testes e consequemente a validações a API URB, executou-se localmente, porém, recursos estão disponibilizados através do servidor do Laboratório de LORDI na UERN, pode ser usado através do navegador por meio do lordi.uern.br:60020/, seguido do serviço que o cliente deseja requisitar assim a API torna acessível para ser consumida pelas aplicações.

Tendo em vista o modelo de Richardson, o qual para uma API ser RESTful deve alcançar a glória REST, a API URB atinge parcialmente os níveis proposto pelo mesmo. Conforme a Figura 8 na seção 2 mostra os níveis, temos que nesta API os recursos são invocados através do mecanismo do protocolo transporte HTTP baseado na invocação de procedimento remoto assim em conformidade com o nível 0. Além do mais, dispõe de rotas que serão expostas posteriormente assim atendendo o nível 1. O cliente pode realizar um CRUD através dos verbos HTTP como será mostrado posteriormente para criar um Reporte faz-se um POST assim está de acordo com o nível 2.

4.3 Validação Web e Mobile

Como plataforma distribuída, a API URB proporciona e disponibiliza os recursos necessários para a criação de aplicações que necessitam do armazenamento e gerenciamento dos problemas urbanos. Um exemplo de sua utilização como fornecedora de serviços pode ser vista na aplicação Reporte Cidadão, que se baseia no consumo dos recursos disponibilizados pelas rotas da API.

4.3.1 Reporte Cidadão

O sistema de contribuição urbana Reporte Cidadão é um conjunto de aplicações e interfaces para o registro e gerenciamento de problemas urbanos. Possibilita que os Cidadãos contribuam através de solicitações sobre os problemas urbanos da sua cidade, podem criar um reporte, um relatório que contém um problema e o organiza em uma ampla categoria. Também permite o gerenciamento destes pelos órgãos gestores responsáveis pela gestão urbana e realizar serviços de infraestrutura. Entre suas interfaces dispõe de um sistema mobile, destinado ao registro e acompanhamento dos reportes pelo usuário reportante, no caso os cidadãos. Além do mais, fornece uma interface web, que auxilia os gestores das cidades a acompanhar as solicitações de serviços públicos e a administrar os problemas urbanos reportados proporcionando um meio de interação com o cidadão ao retornar uma notificação sobre o status do seu relato.

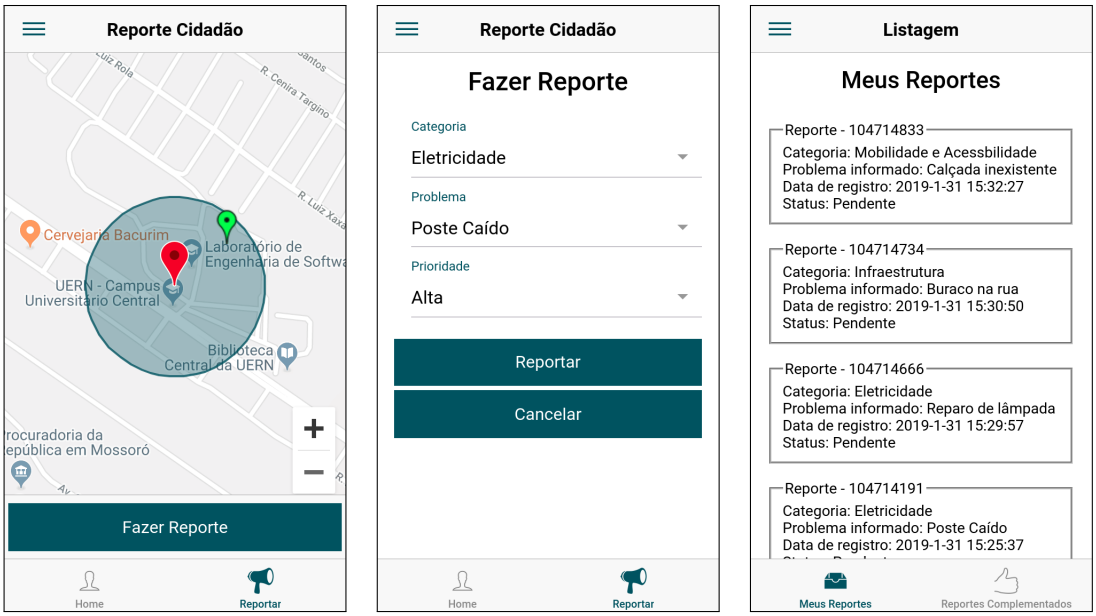
4.3.2 Reporte Cidadão Mobile

A interface do sistema é construída sobre o Ionic Framework e Cordova, possibilitando a utilização da mesma em diferentes plataformas móveis sem a necessidade de recompilação do projeto para diferentes arquiteturas. É responsável por prover funcionalidades ao cidadão para a realização dos relatos dos problemas urbanos encontrados.

Na Figura 31a temos a tela de reportes, nesta após fornecer a localização atual do usuário a API URB é possível listar todos os reportes próximos dentro de um raio específico. A Figura 31b mostra o formulário de Reporte, onde os atributos Categoria e Problema são alimentados pela API segundo a taxonomia proposta neste trabalho. Na Figura 31c a listagem dos reportes feitos pelo cidadão autenticado na aplicação, para mais, é necessário fornecer o identificador único, o id, do usuário para que a API possa processar e filtrar, e possibilitar a aplicação exibir os reportes cadastrados na base de dados realizado pelo o usuário.

A conexão entre o sistema mobile e a API URB pode ser vista no Código 4. Observa-se que a interface acessa o recurso /Reporte, presente no endereço http://localhost:60020, por meio do método POST do protocolo HTTP, vê-se que durante a tentativa de acesso uma variável é enviada, essa consiste das informações do formulário de reporte.

Figura 31 – Aplicação mobile Reporte Cidadão.



(a) (b) (c)

Fonte – Autoria própria.

Código 4 – Código de conexão entre a aplicação mobile e a API URB.

```
this.http.post("http://localhost:60020/Reporte", this.reporteDados).
    subscribe(
  data => {
    const response = data["_body"];
    const objeto_retorno = JSON.parse(response);
  }, erro => {
    console.log("Erro ==> " + erro);
  }
);
```

Fonte – Autoria própria.

4.3.3 Reporte Cidadão Cliente Web

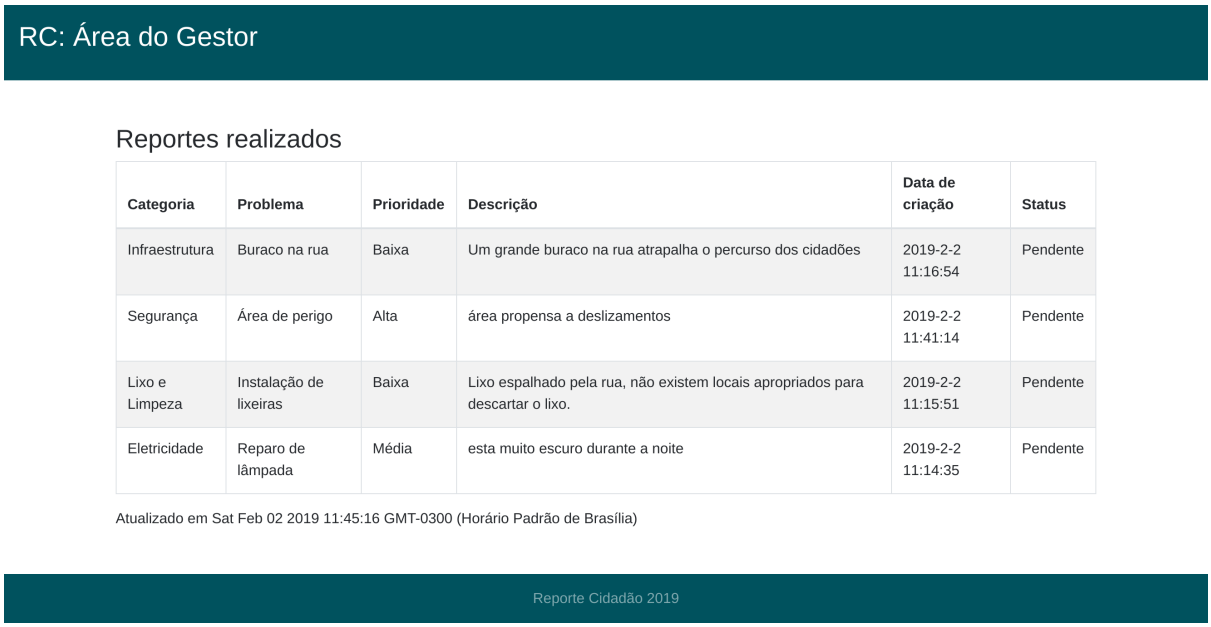
O Reporte Cidadão é formado também de uma interface web, implementada com o Angular, um *Framework* de desenvolvimento web. Esta constituindo-se como uma *Simple Page Aplication* (SPA) uma aplicação que modifica seu comportamento de forma dinâmica.

O cliente web do Reporte Cidadão é responsável por auxiliar os gestores no acompanhamento e gerenciamento dos relatos. Nele são implementados recursos e funcionalizadas para a criação e tratamento das demandas geradas pelos relatos. A Figura 32 traz a tela de listagem de reportes que pertencem ao gestor longado.

O trecho de código Código 5 mostra um exemplo de acesso ao recurso /Reporte,

localizado no endereço http://localhost:60020. Mostra que há a utilização do método GET do protocolo HTTP. Durante esse acesso um objeto JSON é retornado a interface web como resposta a requisição HTTP realizada pelo cliente web.

Figura 32 – Tela de listagem de reportes da Área do Gestor no Cliente Web da aplicação Reporte Cidadão.



Fonte – Autoria própria.

Código 5 – Código de conexão entre a aplicação web e a API URB.

```
listarReportes() {                                                    1
    this.http.get<any[]>(`${'http://localhost:60020/Reporte'}`).      2
   subscribe(
      dados => this.reportes = dados                                  3
   );                                                                 4
}                                                                     5
```

Fonte – Autoria própria.

4.4 Documentação da API URB

O desenvolvimento de APIs com o estilo arquitetural REST, requer que seja produzida uma documentação para que os desenvolvedores *front-end* consigam compreender os serviços providos pela mesma. Na documentação deve existir um detalhamento de cada rota,os códigos de estados que uma rota pode retornar. Pois, sem uma documentação torna difícil uma aplicação comunica-se com uma API REST.

Tendo em vista os aspectos supracitados, a documentação da API URB contém as rotas e recursos que foram detalhadas com o uso de tabelas para tornar a compreensão e leitura mais simples, encontra-se no Apêndice A, para facilitar a utilização das aplicações inclusive de outras APIs.

Com a implementação dos princípios REST é possível garantir a interoperabilidade, sendo um dos objetivos propostos neste trabalho, em que uma aplicação desenvolvida em uma linguagem de programação diferente possa utilizar a API sem impasses, por conseguinte facilitando o desenvolvimento de uma aplicação para funcionar de maneira distribuída.

5 Considerações Finais

Em virtude da atual situação das cidades, o surgimento dos problemas urbanos se dá especialmente pelo fato dessas passarem por um processo de expansão de maneira rápida e desordenada, assim como a falta de um planejamento urbano. Esses aspectos prejudicam a qualidade de vida dos cidadãos, surgindo a necessidade de um suporte adequado para a resolução dessas situações. Assim, é fundamental o desempenho do poder público e dos cidadãos, visando o bem comum na gestão urbana.

Os governos municipais e estaduais são responsáveis por fornecer infraestrutura apropriada aos cidadãos, bem como, têm o dever de solucionar os problemas que surgirem. No entanto, somente a ação do poder público não é suficiente, a população também deve dispor de uma parcela de contribuição na gestão urbana. Dentre os meios existentes utilizados para dar sugestões, percebeu-se o uso de aplicativos em rede como uma ótima opção, uma vez que 94,6% dos brasileiros utilizam os celulares para o uso da internet. Além disso, o cidadão não precisará locomover-se até um órgão para solicitar uma solução. Os indivíduos também dispõem de informações a respeito dos problemas a qualquer momento, evitando a ida a postos de atendimentos presenciais e a preocupação em anotar números de telefones, protocolos, endereços etc.

Diante disso, este trabalho tem por objetivo geral o desenvolvimento de uma API para o gerenciamento dos dados referentes aos problemas urbanos. A API URB fornece uma maneira uniforme para acessar as informações armazenadas e permitir que as aplicações, em diferentes linguagens de programação, possam consumir os serviços providos. Ela foi desenvolvida utilizando a linguagem Javascript e a plataforma Node.js com o auxílio do *framework* Express.

O desenvolvimento resultou em uma API REST com implementação dos requisitos idealizados para a aplicação, baseando-se nos padrões definidos no estilo arquitetural REST. O trabalho também permitiu a criação de uma API simples, em que o desenvolvedor, ao visualizar os serviços do sistema, não tem necessidade de conhecer o código fonte, podendo consumi-los através das requisições para uma URI HTTP. Com a padronização dos recursos cria-se um potencial para interoperabilidade, ou seja, comunicar-se com outras aplicações em plataformas distintas, bem como a implementação de novos serviços, gerando assim novas versões. Por fim, a API contribui o para o gerenciamento dos problemas urbanos, uma vez propõe um modelo que pode ser adotado por outras aplicações para realizar comunicação entre os cidadãos e os gestores das cidades.

O modelo foi efetivado através da elaboração de uma taxonomia dos problemas urbanos, explanando, assim, uma forma de categoriza-los. Além disso, a documentação dessa

API sucedeu com o intuito de facilitar o entendimento dos desenvolvedores, viabilizando seu consumo. Ocorreu a validação da API URB através da aplicação Reporte Cidadão, por meio de sua plataforma web e mobile, mostrando o funcionamento esperado na utilização dos serviços da API Proposta. O cliente fez uso dos métodos acessando a URL correspondente às funcionalidades desejadas.

Para a concretização deste estudo adotou-se como metodologia uma revisão bibliográfica, enfatizando os problemas urbanos. Realizou-se a análise de aplicações observando como são relatados os problemas, além da leitura de artigos relacionados à temática.

Diante da metodologia proposta e dos objetivos alcançados, percebe-se o surgimento de algumas limitações no trabalho, ocorridas em virtude do curto prazo para o desenvolvimento do mesmo. Com base nisso, a próxima subseção apresentará algumas melhorias que podem ser realizadas.

5.1 Trabalhos Futuros

Dado o exposto nesta pesquisa, como trabalhos futuros, pretende-se melhorar as funcionalidades da API URB, dando ênfase aos Gestores, que correspondem aos órgãos responsáveis pela gestão das cidades. É preciso realizar um estudo de como funcionam as ouvidorias dessas partições e de como ocorre o acompanhamento das solicitações de reparo de problemas na cidade. Além disso, torna-se imprescindível validar a aplicação Reporte Cidadão em um cenário real, com intuito de verificar o desempenho da API URB e observar a necessidade de novas funcionalidades ser implementadas. Uma proposta é fazer o uso dentro da própria UERN e, após isso, certificar com uma cidade, com a participação dos cidadãos, realizando relatos de problemas urbanos reais.

Ademais, estudar uma maneira da própria API URB desempenhar função de gerar os relatórios conforme os relatos dos cidadãos, afim de contribuir para um melhor acompanhamento por parte dos gestores. Exemplos de relatórios que poderiam ser gerados são: quantidade de problemas relatados, o tempo gasto para realizar o reparo desses, quais os principais problemas enfrentados e a de quantidade problemas resolvidos.

Outro aspecto, é realizar o processo de mineração dos dados das informações adquiridas por meio dos reportados. Isso possibilitará encontrar padrões, isto é, identificar na cidade a região que tem mais buracos nas vias, por exemplo, além de filtrar as informações de relevância, visando auxiliar os Gestores na tomada de decisão.

Também é preciso refinar cada vez mais a taxonomia URB e fazer outros tipos de classificação, além definir a prioridade dos problemas, categorizando-os como de alta, média e baixa. Alguns problemas requisitam rapidez nos reparos, a exemplo: um poste caído, devendo ser considerado de prioridade alta, pois pode causar a falta de energia e,

em alguns casos, representar risco à segurança dos cidadãos; diferente de uma calçada desnivelada que pode aguardar um intervalo de tempo para ser ajustada, sendo, portanto, de prioridade baixa.

Por fim, contratar ou criar um servidor para realizar a hospedagem da API URB e do banco de dados, porque, no momento, encontra-se disponibilizada apenas por meio do servidor da UERN. Posteriormente, ao obter-se um domínio, pretende-se criar um site para fins informativos e de divulgação, mostrando a taxonomia URB proposta neste trabalho como um modelo para uniformizar a maneira de relatar os problemas urbanos. Além disso, promover os serviços fornecidos pela API URB, bem como dispor da documentação online da mesma, para que, assim, outras aplicações possam utilizá-los.

Referências

ADELIN. *SERVICE ORIENTED ARCHITECTURE - SOA*. 2019. Acessado em 2019. Disponível em: <http://thecoderider.com/service-oriented-architecture-soa/>.

ALMEIDA, E.; GIACOMINI, L. B.; BORTOLUZZI, M. G. Mobilidade e acessibilidade urbana–. *Seminário Nacional de Construções*, 2013.

ALVAREZ, G. M.; CECI, F.; GONÇALVES, A. L. Análise comparativa dos bancos orientados a grafos de primeira e segunda geração–uma aplicação na análise social. *CEP*, v. 88040, p. 900, 2015.

ARANGODB. *ArangoDB Documentation*. 2019. Acessado em 2019. Disponível em: <https://docs.arangodb.com/3.3/Manual/index.html>.

ARAÚJO, D. L. Estudo de uma arquitetura orientada a serviços aplicada a uma plataforma para mineração de dados. 2011.

AURÉLIO. *Developer Survey Results 2018*. 2019. Acessado em 2019. Disponível em: <https://dicionariodoaurelio.com/servico>.

BARROS, D. A. F. Util: Taxonomia unificada para vizualização de informaçã. 2015.

BARRY, D. K. *Web services, service-oriented architectures, and cloud computing*. [S.l.]: Elsevier, 2003.

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

CHOURABI, H. et al. Understanding smart cities: An integrative framework. In: IEEE. *System Science (HICSS), 2012 45th Hawaii International Conference on*. [S.l.], 2012. p. 2289–2297.

CLEZAR, B. O perfil da infra-estrutura urbana das cidades do litoral norte do rio grande do sul. 2006.

COSTA, B. e. a. Evaluating a representational state transfer (rest) architecture: What is the impact of rest in my architecture in: Ieee. software architecture (wicsa). p. 105–114, 2014.

CRUZ, A. L. C. V. D. Soa: concepts and technologies used in service oriented architeture. Intellectus, n. 8, p. 102–121, 2010.

CRUZ, R. *Implementando OAuth JSON Web Token com OWIN no ASP.NET Web AP*. 2019. Acessado em 2019. Disponível em: <https://rafaelcruz.azurewebsites.net/2016/05/13/implementando-oauth-json-web-token-no-asp-net-web-api-e-owin/>.

DIANA, M. D.; GEROSA, M. Um estudo comparativo de bancos não-relacionais para armazenamento de dados na web 2.0. In: *Proceeding of the WTDBD IX Workshop de Teses e Dissertações em Banco de Dados. Belo Horizonte, Brazil: WTDBD*. [S.l.: s.n.], 2010.

DIAS, E. *Desmistificando REST com Java*. [S.l.]: AlgoWorks, 2016.

DINIZ, e. a. AnÁlise dos problemas urbanos do centro da cidade de jacareÍ. p. 5–6, 2011.

DUVANDER, A. *1 in 5 APIs Say "Bye XML", in Programmable Web 2011*. 2011. Acessado em 2018. Disponível em: <https://www.programmableweb.com/news/1-5-apis-say-bye-xml/2011/05/25>.

ERL, T. *SOA: Princípios do design de serviço*. [S.l.]: Pearson Prentice Hall, 2009.

FERREIRA, D. C. Apisim - uma api restful para o gerenciamento de recursos de sistemas operacionais. 2017.

FERREIRA, W. O.; KNOP, I. de O. Estruturação de aplicações distribuídas com a arquitetura rest. *Caderno de Estudos em Sistemas de Informação*, v. 3, n. 1, 2017.

FIELDING, R. T.; TAYLOR, R. N. *Architectural styles and the design of network-based software architectures*. [S.l.]: University of California, Irvine Irvine, USA, 2000. v. 7.

FOWLER, M. *Patterns of enterprise application architecture*. [S.l.]: Addison-Wesley Longman Publishing Co., Inc., 2002.

FOWLER, S. e. *NoSQL distilled: a brief guide to the emerging world of polyglot persistence*. [S.l.]: Pearson Education, 2013.

FURTADO, C. et al. Arquitetura orientada a serviço-conceituação. *Relatórios Técnicos do DIA/UNIRIO (RelaTe-DIA), RT-0012/2009*, 2009.

GOMES, D. A. Web services soap em java: Guia prático para o desenvolvimento de web services java. *Novatec, 2ª edição*, 2014.

GOOGLE. *Soap Api e REST API*. 2019. Acessado em 2019. Disponível em: <http://www.google.com/trends/explore?hl=en-US#q=soap+api,+rest+api&cmpt=q>.

GOOGLE, P. *Pelas Ruas*. 2019. Acessado em 2019. Disponível em: <https://play.google.com/store/apps/details?id=br.com.gruporbs.pelasruas>.

GOOGLE, P. *Problemas Urbanos*. 2019. Acessado em 2019. Disponível em: <https://play.google.com/store/apps/details?id=com.ionicframework.problemasurbanos595575&hl=pt_PT>.

GOOGLE, P. *SP156*. 2019. Acessado em 2019. Disponível em: <https://play.google.com/store/apps/details?id=com.pmsp.sp156>.

GRIGORIK, I. *Armazemar HTTP em cache*. 2019. Acessado em 2019. Disponível em: <https://developers.google.com/web/fundamentals/performance/optimizing-content-efficiency/http-caching?hl=pt-br>.

HARRISON, C.; DONNELLY, I. A. A theory of smart cities. In: *Proceedings of the 55th Annual Meeting of the ISSS-2011, Hull, UK*. [S.l.: s.n.], 2011. v. 55, n. 1.

HAVERBEKE, M. *Eloquent javascript: A modern introduction to programming*. [S.l.]: No Starch Press, 2014.

HORIZONTE, B. Anais da iv conferência de política urbana. 2015.

IBGE. Instituto brasileiro de geografia e estatÍstica. 2010.

IEEE. *Standards Glossary*. 2016. Acessado em 2018. Disponível em: <https://www.ieee.org/education_careers/education/standards/standards_glossary.html>.

JAISWAL, G. Comparative analysis of relational and graph databases. *IOSR Journal of Engineering*, v. 03, p. 25–27, 08 2013.

JOSUTTIS, N. M. *SOA in practice: the art of distributed system design*. [S.l.]: "O'Reilly Media, Inc.", 2007.

KUROSE, J. F.; ROSS, K. W. *Redes de Computadores e a Internet*. [S.l.: s.n.], 2013.

LAURENT, S. S. et al. *Programming Web Services with XML-RPC: Creating Web Application Gateways*. [S.l.]: "O'Reilly Media, Inc.", 2001.

LIMA, J. C. de; CARVALHO, C. L. de. Extensible markup language (xml). 2005.

MAIER, M. W.; EMERY, D.; HILLIARD, R. Ansi/ieee 1471 and systems engineering. *Systems Engineering*, Wiley Online Library, v. 7, n. 3, p. 257–270, 2004.

MARDAN, A. *Express. js Guide: The Comprehensive Book on Express. js*. [S.l.]: Azat Mardan, 2014.

MARQUES, A. I. A. *Desenvolvimento de API para aplicação cloud*. Tese (Doutorado), 2018.

MATOS, M. I. C. d. *Arquitetura de serviços web rest para domótica habitacional*. Dissertação (Mestrado) — Universidade de Aveiro, 2013.

MOCHIZUKI, P. S.; BRESSANE, A.; SALVADOR, N. N. B. Diagnóstico de problemas ambientais urbanos por análises de ocorrências registradas pela população com uso de sistema de informações geográficas em rio claro/sp. *INGEPRO-Inovação, Gestão e Produção*, v. 2, n. 6, p. 019–029, 2010.

MORAES, M. C. de. *A Contribuição da Tecnologia Computacional Inteligente na Gestão da Produção da Energia Elétrica Utilizando Potencial Eólico*. Tese (Doutorado) — PUC-Rio, 2011.

MORATO, R. G. M. et al. Mapeamento da qualidade de vida urbana no município de osasco/sp. *Anais do III Encontro da Associação Nacional de Pós Graduação e Pesquisa em Ambiente e Sociedade. Brasília-Distrito Federal–Brasil*, 2006.

MOSSORÓ. *Plano Diretor de Mossoró*. 2019. Acessado em 2019. Disponível em: <http://www.secovirn.com.br/legislacao/plano-diretor-de-.pdf>.

MÜLLER, I. et al. A conceptual framework for unified and comprehensive soa management. In: SPRINGER. *International Conference on Service-Oriented Computing*. [S.l.], 2008. p. 28–40.

MUSSER, J. Open apis: state of the market. In: *The Glue Conference*. [S.l.: s.n.], 2011.

OLIVEIRA, J. A.; RAMOS, F. G.; JUNIOR, J. J. L. D. Reusabilidade em soa: Um mapeamento sistemático da literatura. 2014.

OLIVEIRA, G. G. D. O. Construção aplicações distribu/idas utilizando-se apis rest. 2018.

OVERFLOW, S. *Developer Survey Results 2018*. 2019. Acessado em 2019. Disponível em: <https://insights.stackoverflow.com/survey/2018>.

PELECHANO, V. et al. Systematic reuse of web services through software product line engineering. In: IEEE. *2011 Ninth IEEE European Conference on Web Services*. [S.l.], 2011. p. 192–199.

PEREIRA, A. J. d. S. Plano de implantação de uma arquitetura orientada a serviços-soa-na câmara dos deputados. 2012.

PEREIRA, R. C. *Construindo APIs REST com Node.js*. [S.l.]: Casa do Codigo, 2016.

PRESSMAN, R. S. *Software engineering: a practitioner's approach*. [S.l.]: Palgrave Macmillan, 2005.

PULUCENO, T. V. Estudo de caso sobre uma api rest utilizando a abordagem de programação orientada e eventos com a plataforma node.js. 2012.

REZENDE, D. A.; ULTRAMARI, C. Plano diretor e planejamento estratégico municipal: introdução teórico-conceitual. *Revista de Administração Pública*, SciELO Brasil, v. 41, n. 2, p. 255–272, 2007.

RIBEIRO, H.; VARGAS, H. C. Urbanização, globalização e saúde. *Revista USP*, n. 107, p. 13–26, 2015.

RIBEIRO, M.; FRANCISCO, R. *WEB SERVICES REST CONCEITOS, ANÁLISE E IMPLEMENTAÇÃO*. [S.l.]: n, 2016.

RICHARDSON. *RESTful Web APIs: Services for a Changing World*. [S.l.]: "O'Reilly Media, Inc.", 2013.

ROBINSON, I.; WEBBER, J.; EIFREM, E. *Graph databases*. [S.l.]: "O'Reilly Media, Inc.", 2013.

RUNSCOPE, A. M. . T. *HTTP Status Codes*. 2019. Acessado em 2019. Disponível em: <https://httpstatuses.com/>.

SAMPAIO, C. *SOA e Web services em Java*. [S.l.]: Brasport, 2006.

SANTOS, M. *A urbanização brasileira*. [S.l.]: Edusp, 2005. v. 6.

SCHAFFERS, H. et al. Smart cities and the future internet: Towards cooperation frameworks for open innovation. In: SPRINGER. *The future internet assembly*. [S.l.], 2011. p. 431–446.

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

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

SILVA, H.; MONTE-MÓR, R. L. Transições demográficas, transição urbana, urbanização extensiva: um ensaio sobre diálogos possíveis. *Anais*, p. 1–16, 2016.

SOMMERVILLE, I. *Software engineering*. [S.l.: s.n.], 2011. v. 137035152.

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

SOUZA, M. J. L.; RODRIGUES, G. B. *Planejamento urbano e ativismos sociais*. [S.l.]: Unesp, 2004.

STAL, M. Using architectural patterns and blueprints for service-oriented architecture. *IEEE software*, IEEE, v. 23, n. 2, p. 54–61, 2006.

STERVINOU, J. *XML-RPC.Com*. 2019. Acessado em 2019. Disponível em: <http://xmlrpc.scripting.com/>.

TOBALDINI, R. Arquitetura rest: um estudo de sua implementação em linguagens de programação. 2007. *Monografia. UNIVERSIDADE FEDERAL DE SANTA CATARINA, Florianópolis*, 2007.

W3C. *World Wide Web Consortium*. 2019. Acessado em 2019. Disponível em: <https://www.w3.org/TR/ws-arch/>.

APÊNDICE A – Rotas

Neste apêndice apresenta todas rotas que foram implementadas para a API URB.

Tabela 3 – Detalhes da rota para criar um Solver.

URI	/solver
Método	POST.
Parâmetros	Nenhum.
Corpo	nomeSolver

Fonte – Autoria própria.

Tabela 4 – Detalhes para criar uma Categoria.

URI	/categorias
Método	POST.
Parâmetros	Nenhum.
Corpo	nomeCategoria

Fonte – Autoria própria.

Tabela 5 – Detalhes para criar um Problema.

URI	/problemas
Método	POST.
Parâmetros	Nenhum.
Corpo	nomeProblema

Fonte – Autoria própria.

Tabela 6 – Detalhes para criar um Cidadão.

URI	/cidadaos
Método	POST.
Parâmetros	Nenhum.
Corpo	nomeCidadao, senhaCidadao ,emailCidadao

Fonte – Autoria própria.

Tabela 7 – Detalhes para criar um Gestor.

URI	/gestores
Método	POST.
Parâmetros	Nenhum.
Corpo	nomeGestor, cnpjGestor, senhaGestor ,emailGestor.

Fonte – Autoria própria.

Tabela 8 – Detalhes para criar um reporte.

URI	/reportes
Método	POST.
Parâmetros	Nenhum.
Corpo	categoria, problema, prioridade, descricaoDoProblema, coordinate, descricaoDoLocal, urlPhoto

Fonte – Autoria própria.

Tabela 9 – Detalhes para criar uma Demanda.

URI	/demandas
Método	POST.
Parâmetros	Nenhum.
Corpo	descricaoDemanda,categoria, problema, coordinate, descricaoDoLocal.

Fonte – Autoria própria.

Tabela 10 – Detalhes para criar uma relação entre Categoria e Problema.

URI	/hasCategoriaProblemas
Método	POST.
Parâmetros	Nenhum.
Corpo	to_id,from_id.

Fonte – Autoria própria.

Tabela 11 – Detalhes para criar uma relação entre Cidadão e Reporte.

URI	/hasCidadaoReporte
Método	POST.
Parâmetros	Nenhum.
Corpo	to_id,from_id.

Fonte – Autoria própria.

Tabela 12 – Detalhes para criar um Solver.

URI	/hasCreateSolver
Método	POST.
Parâmetros	Nenhum.
Corpo	to_id,from_id.

Fonte – Autoria própria.

Tabela 13 – Detalhes para criar uma Demanda e um Reporte.

URI	/hasDemandaReporte
Método	POST.
Parâmetros	Nenhum.
Corpo	to_id,from_id.

Fonte – Autoria própria.

Tabela 14 – Detalhes para criar uma relação Demanda e um Gestor.

URI	/hasDemandaGestor
Método	POST.
Parâmetros	Nenhum.
Corpo	to_id,from_id.

Fonte – Autoria própria.

Tabela 15 – Detalhes para criar uma relação Demanda e um Reporte.

URI	/hasDemandaReporte
Método	POST.
Parâmetros	Nenhum.
Corpo	to_id,from_id.

Fonte – Autoria própria.

Tabela 16 – Detalhes para criar uma relação entre Solver e uma Demanda.

URI	/hasSolverDemanda
Método	POST.
Parâmetros	Nenhum.
Corpo	to_id,from_id.

Fonte – Autoria própria.

Tabela 17 – Detalhes para criar realizar Login.

URI	/Login
Método	POST.
Parâmetros	auth.
Corpo	email, senha.

Fonte – Autoria própria.

Tabela 18 – Rotas do Cidadão.

URI	MÉTODO	DESCRIÇÃO
/cidadados	POST	Cadastrar um Cidadão.
/cidadados	GET	Listar todos os Cidadãos.
/cidadados/:id	GET	Buscar um Cidadão pelo ID.
/cidadados/:id	PUT	Editar os dados de um Cidadão.
/cidadados/:id	DELETE	Deletar um Cidadão.

Fonte – Autoria própria.

Tabela 19 – Rotas do Gestor.

URI	MÉTODO	DESCRIÇÃO
/gestores	POST	Cadastrar um Gestor.
/gestores	GET	Listar todos os Gestores.
/gestores/:id	GET	Buscar um Gestor pelo ID.
/gestores/:id	PUT	Editar os dados de um Gestor.
/gestores/:id	DELETE	Deletar um Gestor.

Fonte – Autoria própria.

Tabela 20 – Rotas do Solver.

URI	MÉTODO	DESCRIÇÃO
/solver	POST	Cadastrar um Solver.
/solver	GET	Listar todos os Solver.
/solver/:id	GET	Buscar um Solver pelo ID.
/solver/:id	PUT	Editar os dados de um Solver.
/solver/:id	DELETE	Deletar um Solver.

Fonte – Autoria própria.

Tabela 21 – Rotas do Reporte.

URI	MÉTODO	DESCRIÇÃO
/reportes	POST	Cadastrar um Reporte.
/reportes	GET	Listar todos os Reporte.
/listar-reporte-pelo-raio/:idReporte	GET	Listar os reportes dentro de um raio pelo ID.
/rerpotes/:id	GET	Buscar um Reporte pelo ID.
/rerpotes/:id	PUT	Editar os dados de um Reporte.
/rerpotes/:id	DELETE	Deletar um Reporte.

Fonte – Autoria própria.

Tabela 22 – Rotas da Demanda.

URI	MÉTODO	DESCRIÇÃO
/demandas	POST	Cadastrar uma Demanda.
/demandas	GET	Listar todos as Demanda.
/demandas/:id	GET	Buscar uma Demanda pelo ID.
/demandas/:id	PUT	Editar os dados de uma Demanda.
/demandas/:id	DELETE	Deletar uma Demanda.

Fonte – Autoria própria.

Tabela 23 – Rotas de Categoria.

URI	MÉTODO	DESCRIÇÃO
/categorias	POST	Cadastrar uma Categoria.
/categorias	GET	Listar todos as Categoria.
/categorias/:id	GET	Buscar uma Categoria pelo ID.
/categorias/:id	PUT	Editar os dados de uma Categoria.
/categorias/:id	DELETE	Deletar uma Categoria.

Fonte – Autoria própria.

Tabela 24 – Rotas de Problema.

URI	MÉTODO	DESCRIÇÃO
/problemas	POST	Cadastrar um Problema.
/problemas	GET	Listar todos os Problemas.
/problemas/:id	GET	Buscar um Problema pelo ID.
/problemas/:id	PUT	Editar os dados de um Problema.
/problemas/:id	DELETE	Deletar um Problema.

Fonte – Autoria própria.

Tabela 25 – Rotas de hasCategoriaProblemas.

URI	MÉTODO	DESCRIÇÃO
/hasCategoriaProblemas	POST	Criar uma relação entre Categoria e Problema.
/hasCategoriaProblemas/:id	GET	Listar os Problemas pelo id de uma Categoria.

Fonte – Autoria própria.

Tabela 26 – Rotas de hasCidadaoReporte.

URI	MÉTODO	DESCRIÇÃO
/hasCidadaoReporte	POST	Criar uma relação entre Cidadão e Reporte.
/hasCidadaoReporte/:id	GET	Listar os Reporte pelo id de um Cidadão.

Fonte – Autoria própria.

Tabela 27 – Rotas de hasDemandaReporte.

URI	MÉTODO	DESCRIÇÃO
/hasDemandaReporte	POST	Criar uma relação entre Demanda e Reporte.
/hasDemandaReporte/:id	GET	Listar as Demanda pelo id de um Reporte.

Fonte – Autoria própria.

Tabela 28 – Rotas de hasDemandaGestor.

URI	MÉTODO	DESCRIÇÃO
/hasDemandaGestor	POST	Criar uma relação entre Demanda e Gestor.
/hasDemandaGestor/:id	GET	Listar as Demanda pelo id de um Gestor.

Fonte – Autoria própria.

Tabela 29 – Rotas de hasSolverDemanda.

URI	MÉTODO	DESCRIÇÃO
/hasSolverDemanda	POST	Criar uma relação entre Solver e Demanda.
/hasSolverDemanda/:id	GET	Listar as Demanda pelo id de um Solver.

Fonte – Autoria própria.

Tabela 30 – Rotas de hasProblemaReporte.

URI	MÉTODO	DESCRIÇÃO
/hasProblemaReporte	POST	Criar uma relação entre Problema e Reporte.
/hasProblemaReporte/:id	GET	Listar os Reporte pelo id de um Problema.

Fonte – Autoria própria.