UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

GABRIEL SOUZA PEPLINSKI

ÁGORA: PARTICIPAÇÃO CIDADÃ NA MANUTENÇÃO URBANA

GUARAPUAVA

2025

GABRIEL SOUZA PEPLINSKI

ÁGORA: PARTICIPAÇÃO CIDADÃ NA MANUTENÇÃO URBANA

Ágora: Citizen Participation in Urban Maintenance

Monografia de Trabalho de Conclusão de Curso de Graduação apresentado à disciplina de Trabalho de Conclusão de Curso 2, do Curso Superior de Tecnologia em Sistemas para Internet da Universidade Tecnológica Federal do Paraná, Campus Guarapuava, como requisito parcial para obtenção do título de Tecnólogo em Tecnologia em Sistemas para Internet.

Orientador: Me. Dênis Lucas Silva

GUARAPUAVA

2025

4.0 Internacional

Esta licença permite compartilhamento, remixe, adaptação e criação a partir do trabalho, mesmo para fins comerciais, desde que sejam atribuídos créditos ao(s) autor(es). Conteúdos elaborados por terceiros, citados e referenciados nesta obra não são cobertos pela licença.

GABRIEL SOUZA PEPLINSKI

ÁGORA: PARTICIPAÇÃO CIDADÃ NA MANUTENÇÃO URBANA

Monografia de Trabalho de Conclusão de Curso de Graduação apresentado à disciplina de Trabalho de Conclusão de Curso 2, do Curso Superior de Tecnologia em Sistemas para Internet da Universidade Tecnológica Federal do Paraná, Campus Guarapuava, como requisito parcial para obtenção do título de Tecnólogo em Tecnologia em Sistemas para Internet.

Data de aprovação: 19/fevereiro/2025

William Alberto Cruz Castaneda
Doutor
Universidade Tecnológica Federal do Paraná

Renata Luiza Stange
Doutora
Universidade Tecnológica Federal do Paraná

Dênis Lucas Silva
Mestre
Universidade Tecnológica Federal do Paraná

GUARAPUAVA
2025

RESUMO

Nos ambientes urbanos, muitas vezes, o cidadão se depara com problemas relacionados à cidade onde reside, seja por falta de planejamento prévio ou por outro motivo, sempre existem diferentes tipos de ocorrências que afetam o dia a dia dos seus habitantes, reduzindo e impactando a qualidade de vida dessas pessoas. Este trabalho propõe o desenvolvimento de um sistema onde o próprio cidadão, por meio de seu celular, realiza o registro dessas ocorrências que o afetam. Além disso, outros usuários também poderão ter acesso a esses dados e acompanhar informações como: quais os problemas mais comuns, quais os locais mais afetados e o que está sendo realizado para resolver cada um. Com a implementação deste sistema, o objetivo é fazer o cidadão ocupar seu lugar na sociedade e exercer seu papel de forma ativa, como um agente fiscalizador dentro do ambiente onde está inserido.

Palavras-chave: aplicativo móvel; participação cidadã; problemas urbanos; api rest.

ABSTRACT

In urban environments, citizens often encounter problems related to the city in which they live, whether due to lack of prior planning or for other reasons. There are always different types of occurrences that affect the daily lives of its inhabitants, reducing and impacting the quality of life for these people. This work aims to propose the development of a system that the citizen themselves, through a cell phone, can register occurrences that affect his or her life; furthermore, other users will also be able to access this data and follow information such as: the most common problems, the most affected places, and what is being done to resolve each one. With the implementation of this system, the main objective is to make citizens assume their role in society and play an active part as monitoring agents within their surroundings.

Keywords: mobile application; citizen participation; urban problems; rest api.

LISTA DE FIGURAS

Figura 1 – Captura de tela da interface do site Carioca Digital mostrando o painel, onde contém um relatório geral das solicitações registradas e atendidas por categoria 15

Figura 2 – Captura de tela da interface do aplicativo Atende Net mostrando a sua tela principal 16

Figura 3 – Metodologia de desenvolvimento 21

Figura 4 – Fluxo de desenvolvimento. 23

Figura 5 – Fluxo de Ramificações 24

Figura 6 – Diagrama MER do sistema 28

Figura 7 – Arquivo *docker-compose.yml*, com as configurações de imagens e contêineres Docker para o ambiente de desenvolvimento do sistema Ágora. 34

Figura 8 – Arquivo de comandos de instalação e configurações iniciais, utilizado para configurar o ambiente pela primeira vez. 35

Figura 9 – Estrutura de diretórios baseada no DDD utilizada na API e diagrama para demonstrar de forma mais lúdica esta organização. 35

Figura 10 – Arquivo de caso de testes unitários. 36

Figura 11 – Arquivo de configurações de estilização de código para a ferramenta Pint. 36

Figura 12 – Comando de execução de verificação e correção de estilização de código PHP utilizando Pint. 37

Figura 13 – Workflow para verificar a estilização de código PHP com o uso do Pint. 37

Figura 14 – Exemplo de execução correta de workflows no GitHub. 37

Figura 15 – Exemplo de contexto em React, demonstrando a importação do Provider da biblioteca React Native Paper no layout básico da aplicação, de forma que, os componentes dessa biblioteca possam ser chamados em todos os lugares da aplicação. 38

Figura 16 – Instância do Axios com configurações padrão utilizadas no sistema. . 39

Figura 17 – Capturas de telas que compõem a autenticação do sistema: Tela de login e cadastro de usuário. 41

Figura 18 – Captura de tela de aplicativo da tela de listagem de solicitações do usuário atual e tela de exibição de dados de solicitação de forma individual. **42**
Figura 19 – Captura de tela de aplicativo efetuando a atualização de situação e captura de tela exibindo detalhes de uma solicitação. **43**
Figura 20 – Captura de tela de aplicativo da tela endereço e atualização de dados pessoais. **44**
Figura 21 – Captura de tela de aplicativo pedindo permissão para acesso de localização em tempo real e tela de mapa após a permissão concedida. . . . **45**
Figura 22 – Captura de tela de aplicativo, exibindo formulário de cadastro de nova solicitação de manutenção urbana e processo de captura de foto via câmera. **46**
Figura 23 – Captura de tela do aplicativo exibindo uma solicitação reforçada com sucesso. O ícone de reforço está destacado, indicando visualmente ao usuário que a solicitação já recebeu seu apoio. **47**
Figura 24 – Imagem lúdica, com captura de tela de mapa do OpenStreetMap, demonstrando a importância da participação dos cidadãos, pois cada um pode ser um agente fiscalizador no local onde vive. **49**
Figura 25 – Store criado utilizando o Zustand juntamente com o AsyncStorage. . . **54**
Figura 26 – Classe que implementa as validações Yup para validar dados de entrada para o login. **55**
Figura 27 – Formulário de login utilizando as bibliotecas Formik e Yup. **56**
Figura 28 – Diagrama do banco de dados do sistema, exibindo as tabelas em sua primeira versão, de acordo com o MER produzido como resultado na análise inicial e as tabelas abaixo após a atualização. **57**
Figura 29 – Contexto para controle e acesso de localização do dispositivo móvel via GPS. **67**
Figura 30 – Request para validação de dados de entrada para cadastro/edição de solicitações de manutenção urbana. **68**
Figura 31 – Data Transfer Object (DTO) para dados de solicitação de manutenção urbana. **69**
Figura 32 – Action para cadastro de nova solicitação de manutenção urbana. . . . **70**

Figura 33 – Teste unitário desenvolvido para testar execução da Action para cadastro de nova solicitação de manutenção urbana. **71**
Figura 34 – Service para cadastro de novas solicitações de manutenção urbanas, utilizado no aplicativo. **71**
Figura 35 – Classe base de upload de imagens utilizada na aplicação. **72**
Figura 36 – Exemplo de modelo configurado para poder usar imagens com a biblioteca laravel-medialibrary. **73**
Figura 37 – Exemplo de vinculação de imagem a modelo utilizando o pacote laravel-medialibrary. **73**
Figura 38 – Comando personalizado para atualização de perfis e permissões através do terminal da aplicação com o uso do Artisan. **74**
Figura 39 – Diagrama de tabelas que correspondem ao armazenamento de dados de endereço dos usuários do sistema. **76**
Figura 40 – Diagrama de tabelas que correspondem ao armazenamento de dados de usuários e solicitações. **78**
Figura 41 – Diagrama de tabelas que correspondem ao armazenamento de perfis e permissões do sistema. **79**

LISTA DE ABREVIATURAS E SIGLAS

Siglas

API	Application Programming Interface
CLI	Command-Line Interface
DDD	*Domain-Driven Design*
DTO	Data Transfer Object
GPS	Sistema de Posicionamento Global, do inglês Global Positioning System
MER	Modelo Entidade Relacionamento
NPM	Node Package Manager
ORM	*Object-Relational Mapping*
PHP	Hypertext Preprocessor
SGBD	Sistema de Gerenciamento de Banco de Dados
SPA	Single Page Applications
TDD	Test Driven Development
YAML	YAML Ain't Markup Language

SUMÁRIO

1 INTRODUÇÃO 10

1.1 Considerações iniciais 10

1.2 Objetivos 12

1.2.1 Objetivo geral 12

1.2.2 Objetivos específicos 12

1.3 Justificativa 13

1.4 Estrutura do trabalho 14

2 SISTEMAS SIMILARES 15

3 SISTEMA ÁGORA 17

3.1 Escopo 17

3.2 Perspectivas 17

4 MATERIAIS E MÉTODOS 19

4.1 Materiais 19

4.2 Métodos 21

5 ANÁLISE DO SISTEMA 26

5.1 Lista de Requisitos 26

5.2 Modelo Entidade Relacionamento (MER) 27

6 AMBIENTE DE DESENVOLVIMENTO 29

7 RESULTADOS 40

7.1 Telas da Aba Área de Usuário 40

7.2 Telas da Aba Mapa 43

8 CONCLUSÕES 48

8.1 Trabalhos Futuros 48

REFERÊNCIAS 50

APÊNDICE A DESENVOLVIMENTO DO SISTEMA ÁGORA 53

A.1 Ciclo 1 53

A.1.1 Autenticação de Usuários 53

A.2 Ciclo 2 56

A.2.1 Cadastro de Endereço de Usuários 56

A.2.2 Evolução da Estrutura do Banco de Dados 57
A.3 Ciclo 3 58
A.3.1 Integração com o *GPS* para Captura de Localização Atual 58
A.3.2 Problemas de Compatibilidade 58
A.4 Ciclo 4 59
A.4.1 Registro de Categorias através de Seeders 59
A.4.2 Cadastro de Solicitações de Manutenção Urbanas 59
A.4.3 Captura de Imagens via Aplicativo Mobile e Envio para a API 60
A.4.4 Gerenciamento de Imagens na API 61
A.4.5 Reforço de Solicitação de Manutenção Urbana 62
A.5 Ciclo 5 62
A.5.1 Atualização de Dados Pessoais e de Acesso 63
A.5.2 Sistema de Perfis e Permissões 63
A.5.3 Exclusão de Solicitações de Manutenção Urbana 64
A.6 Ciclo 6 64
A.6.1 Atualização de Situação da Solicitações de Manutenção Urbana 64
A.6.2 Delimitação de Área de Atuação do Usuário 65
A.7 Ciclo 7 65
A.7.1 Edição de Solicitações de Manutenção Urbana 65
A.7.2 Melhorias e Ajustes Finais do Aplicativo e API 66
APÊNDICE B MODELAGEM DO BANCO DE DADOS DO SISTEMA . . 76
B.0.1 Tabelas de Endereço 76
B.0.2 Tabelas de Usuários e Solicitações 76
B.0.3 Tabelas de Perfis e Permissões 77

1 INTRODUÇÃO

Nesse capítulo será apresentada uma visão geral sobre o trabalho desenvolvido, mostrando o cenário onde ele se aplica, bem como a motivação e as justificativas do seu desenvolvimento. Além disso, os objetivos que precisam ser cumpridos para que o trabalho seja concluído com êxito.

1.1 Considerações iniciais

A qualidade de vida pode ser determinada pela observação de uma série de fatores na vida de uma pessoa, um conjunto de condições que contribuem para o seu bem-estar. Sendo assim, pode incluir, por exemplo, autoestima, finanças, relações pessoais e satisfação com atividades rotineiras. Porém, é difícil medir, de forma geral, a qualidade de vida de uma pessoa, pois é preciso entender quais fatores ela considera em sua vida para determinar se possui ou não qualidade de vida.

Contudo, é possível dizer que alguns aspectos são comuns a todas as pessoas. Aspectos profissionais e relacionados à saúde são exemplos. Também os aspectos sociais, que estão além das relações com outras pessoas, fazendo com que o meio ambiente, ou seja, tudo o que está ao redor, tenha impacto no bem-estar geral das pessoas.Como mencionado por BemEstar (2025), viver em uma cidade com excesso de lixo, poluição e falta de áreas verdes pode contribuir significativamente para o desenvolvimento de doenças e para a redução do bem-estar e da qualidade de vida (BEMESTAR, 2025).

Todas as cidades, independentemente de seu tamanho, enfrentam desafios que afetam seus cidadãos diariamente. Entre os problemas mais comuns enfrentados no dia a dia da maioria dos residentes de áreas urbanas, dentre muitos, de acordo com Jacobi (2013), podem ser citados:

a) Problemas de mobilidade: ruas em condições precárias, falta de sinalização em ruas ou avenidas perigosas e movimentadas, problemas relacionados à acessibilidade de pessoas com deficiência ou mobilidade reduzida (cadeirantes ou pessoas com algum grau de deficiência visual), por exemplo, falta de rampas e de calçamento com piso tátil direcional;

b) Problemas relacionados ao ambiente urbano: ruas sem iluminação, ambientes públicos em más condições de preservação (falta de manutenção em áreas gramadas como parques e descarte indevido de lixo e resíduos);

c) Problemas relacionados à qualidade de vida: falta de saneamento básico e moradia em áreas impróprias para a habitação.

De acordo com Marques (2025), a paisagem urbana está em constante crescimento e transformação, à medida que um número cada vez maior de pessoas opta por viver e trabalhar em áreas urbanas. No entanto, o autor ressalta que, infelizmente, esse processo raramente ocorre de maneira planejada e estruturada, especialmente em países mais pobres, o que contribui para o surgimento de problemas urbanos. Independentemente de quem são os causadores diretos de cada um dos problemas encontrados nas cidades, é preciso que o poder público, neste caso, as prefeituras, se antecipem aos incômodos causados e comecem a agir. Contudo, pode-se imaginar que tomar conhecimento sobre os problemas de uma cidade não é uma tarefa fácil, principalmente se a cidade é grande e populosa. Para contornar esta situação, muitas prefeituras têm estabelecido canais para abordar esse desafio. Um exemplo disso são os portais de atendimento digital, como o Portal Carioca Digital, o SP156 e, em Guarapuava, o Atende Net.

Dois grandes problemas podem ser notados neste cenário descrito até o momento: primeiro, a sensação do cidadão de que um problema encontrado por ele demorará para chegar ao conhecimento das autoridades municipais; segundo, que os canais disponibilizados para fazer denúncias sobre um problema não funcionam e/ou não são transparentes. Essa falta de transparência faz com que o cidadão fique em dúvida sobre se um problema já foi informado ou não, resultando, muitas vezes, na invisibilidade do problema, pois ninguém relata, achando que alguém já o fez. Além disso, quando o relato é feito, fica a sensação de que nada será feito a respeito, pois ao passar do tempo, não se vê nenhuma ação por parte do poder público e seus responsáveis.

Consequentemente, as pessoas se acostumam com essas situações adversas sem buscar soluções, muitas vezes por falta de conhecimento sobre a quem recorrer ou por desacreditarem que algo será feito. Porém, em qualquer sociedade, o cidadão desempenha um papel de extrema importância. Além de constituir a sua base, o cidadão detém o poder de escolher seus representantes e, igualmente significativo, atuar como agente fiscalizador. Segundo o artigo de Camp (1986), na Grécia Antiga, considerada o berço da democracia, os cidadãos se reuniam em uma praça pública onde debatiam uma ampla variedade de tópicos, abrangendo desde política, comércio, justiça até administração pública. Essa participação ativa na tomada de decisões e discussões era essencial para cumprir o papel de cidadão. Este lugar era chamado de Ágora (CAMP, 1986). Isto inspira o nome do sistema proposto neste trabalho. A alusão deste nome com a palavra “agora”[1] faz total sentido com o sistema proposto, ou seja, no momento em que cada cidadão encontrar um problema na sua cidade, seja uma calçada com buracos ou um terreno abandonado servindo de depósito de lixo, ele poderá relatá-lo.

O maior diferencial do Ágora frente aos sistemas similares mencionados anteriormente, é que todos os relatos ficarão públicos. Desta forma, estarão acessíveis, aumentando a transparência desse processo. Além disso, o sistema Ágora almeja fomentar, por meio do seu uso, a participação da sociedade no desenvolvimento urbano, sendo este, inclusive, um dos oito objetivos estratégicos da Carta Brasileira para Cidades Inteligentes, emitida em 2020:

[1] Nesta hora; neste exato momento; já (https://www.dicio.com.br/agora/).

> Objetivo Estratégico 7: Fomentar um movimento massivo e inovador de educação e comunicação públicas para maior engajamento da sociedade no processo de transformação digital e de desenvolvimento urbano sustentáveis (CARTA. . . , 2020).

Também, na Primeira Conferência Internacional sobre Promoção da Saúde, realizada em Ottawa, Canadá, em novembro de 1986, os princípios fundamentais da promoção da saúde e as diretrizes para ações e políticas para melhorar a saúde das populações foram discutidos (EUROPE, 1986). Logo, enquanto política pública de promoção da saúde, para melhorar a qualidade de vida da população, é preciso envolver as pessoas no cuidado e zelo das cidades.

> Para atingir um estado de completo bem-estar físico, mental e social os indivíduos e grupos devem saber identificar aspirações, satisfazer necessidades e modificar favoravelmente o meio ambiente. A saúde deve ser vista como um recurso para a vida, e não como objetivo de viver. Nesse sentido, a saúde é um conceito positivo, que enfatiza os recursos sociais e pessoais, bem como as capacidades físicas. Assim, a promoção da saúde não é responsabilidade exclusiva do setor saúde, e vai para além de um estilo de vida saudável, na direção de um bem estar global (EUROPE, 1986)

A adoção do uso do sistema pode conscientizar o cidadão da sua responsabilidade na manutenção urbana. De forma a entender que ao zelar pelo seu entorno, está criando uma cultura de cuidados, que resulta em um sentimento de pertencimento e, por sua vez, em qualidade de vida. Na seção 3.1 o sistema Ágora é descrito com mais detalhes, apresentando seu escopo e perspectivas.

1.2 Objetivos

Nessa seção serão apresentados os objetivos com o desenvolvimento e implantação do sistema Ágora.

1.2.1 Objetivo geral

Desenvolvimento de um sistema para registro e consultas de solicitações de manutenção urbana.

1.2.2 Objetivos específicos

- Desenvolver um Application Programming Interface (API), com a qual se possa realizar:
  - Cadastro e autenticação de usuários;

  - Operações CRUD para solicitações de manutenção urbanas;

- Desenvolver uma aplicação móvel, com a qual o usuário cidadão possa realizar:
  - Uso de localização atual via GPS do dispositivo móvel;
  - Operações CRUD para solicitações de manutenção urbanas;
  - Funcionalidade de reforço para solicitações já cadastradas;

1.3 Justificativa

Todas as cidades enfrentam problemas no processo de urbanização. Para Vieira (2023), nos centros urbanos, os cidadãos sofrem diariamente com esses problemas, que, muitas vezes, acabam reduzindo sua qualidade de vida ou, em casos extremos, colocando-os em risco. Quando a infraestrutura da cidade não suporta a demanda gerada pelos habitantes, sempre ocorrem problemas urbanos Vieira (2023). Ademais, ser cidadão quer dizer participar ativamente da sociedade.

> Mais do que ser eleitor em apenas um dia, compete ao cidadão acompanhar de forma permanente o que está sendo executado em seu município. A qualidade da gestão está diretamente ligada à prática de fiscalização por parte dos moradores que devem, por sua vez, cobrar dos políticos as promessas de campanha e ações que tragam mais qualidade de vida para a sociedade (Tribunal Superior Eleitoral, 2017).

A possibilidade de registrar situações indesejadas encontradas no seu entorno ou em qualquer outro ponto da cidade, bem como permitir que outros cidadãos tenham acesso a esse tipo de informação de forma transparente, contribui para a sociedade, pois o cidadão poderá cumprir seu papel de agente fiscalizador.

A partir disso, pode-se concluir que, ao disponibilizar o sistema Ágora na forma de aplicativo, ele se tornará mais acessível à população, pois, nos dias de hoje, o acesso à internet e a dispositivos móveis, como celulares, está bem difundido. Segundo uma pesquisa realizada pelo Centro Regional de Estudos para o Desenvolvimento da Sociedade da Informação (Cetic.br) do Núcleo de Informação e Coordenação do Ponto BR (NIC.br)[2], 92 milhões de brasileiros têm acesso à internet somente pelo celular, representando 62% da população. Outra vantagem é a possibilidade de registrar uma solicitação no exato momento em que se encontrar uma situação fora de conformidade, como as citadas anteriormente.

[2] https://nic.br/noticia/releases/92-milhoes-de-brasileiros-acessam-a-internet-apenas-pelo-telefone-celular-aponta-tic-domicilios-2022/

1.4 Estrutura do trabalho

Este trabalho está estruturado da seguinte forma: o Capítulo 2 apresenta sistemas análogos e relaciona suas contribuições ao Ágora. No Capítulo 3, são detalhadas as funcionalidades do sistema, seu funcionamento geral e suas possíveis aplicações. O Capítulo 4 descreve as tecnologias, frameworks, ferramentas e bibliotecas utilizadas no desenvolvimento, além do fluxo de trabalho adotado e do processo de documentação. No Capítulo 5, são analisados os requisitos funcionais e apresentado o Modelo Entidade Relacionamento (MER), que serviu como base para a estruturação do banco de dados. O Capítulo 6 aborda o processo de implantação do ambiente de desenvolvimento, incluindo a configuração dos projetos *Laravel* (para a API) e Expo (para o aplicativo), bem como a criação do ambiente local com a ferramenta Docker. No Capítulo 7, apresenta as telas do aplicativo desenvolvido, juntamente com o fluxo de navegação. Por fim, o Capítulo 8 traz as considerações finais e as perspectivas futuras do projeto.

2 SISTEMAS SIMILARES

Atualmente, existem sistemas com funções análogas ao objetivo do sistema Ágora. Várias prefeituras disponibilizam aplicações nas quais o usuário pode registrar suas solicitações de manutenção urbana. No entanto, a maioria desses sistemas não é focada exclusivamente em atender a essas solicitações, mas sim em ser mais generalizada, agrupando diversas funcionalidades. Em decorrência disso, esses sistemas não possuem muitas funcionalidades voltadas para esse objetivo e, além disso, nenhum apresenta informações de forma transparente ao usuário, como foi previamente mencionado.

Dentre esses sistemas similares, está o Portal 1746[1], acessado através do site Portal Carioca Digital, este sistema é implementado pela Prefeitura da cidade do Rio de Janeiro, que, por meio da ouvidoria, oferece aos cidadãos a oportunidade de registrar suas solicitações de manutenção urbana e acompanhar seu progresso por meio dos números de protocolo. Além disso, o sistema fornece gráficos informativos que detalham a quantidade de reclamações por setor, fornecendo uma visão abrangente da situação. A Figura 1 apresenta a tela deste sistema com a sumarização das solicitações.



Figura 1 – Captura de tela da interface do site Carioca Digital mostrando o painel, onde contém um relatório geral das solicitações registradas e atendidas por categoria
Fonte: Site Portal 1746.

Outra ferramenta semelhante está em operação no município de São Paulo, conhecida como SP156[2], que permite aos moradores registrar solicitações de manutenção urbana e obter respostas de operadores, caso mais informações sejam necessárias, ou para saber sobre os resultados obtidos com as reclamações.

[1] https://www.1746.rio/hc/pt-br
[2] https://sp156.prefeitura.sp.gov.br/portal/servicos

Em Guarapuava, essa funcionalidade está presente também através da ouvidoria municipal, no sistema Atende Net[3]. No entanto, vale ressaltar que este sistema é mais simples em comparação com os sistemas mencionados anteriormente, limitando-se ao acompanhamento das reclamações apenas por meio do número de protocolo.



Figura 2 – Captura de tela da interface do aplicativo Atende Net mostrando a sua tela principal
Fonte: Aplicativo Atende Net.

Todos os três sistemas mencionados possuem uma versão para celular. Porém, nenhum dos aplicativos, de acordo com a loja de aplicativos para dispositivos Android, Google Play, tem uma boa avaliação de seus usuários. Dentre todos, o mais bem avaliado é o Atende Net (Figura 2). Ainda assim, o processo de registrar solicitações em Guarapuava é desconfortável devido a várias limitações do aplicativo e do sistema WEB. No sistema WEB, a interface não é muito amigável, as cores usadas são desagradáveis, e tanto na versão mobile quanto na WEB, os usuários têm apenas algumas categorias e subcategorias limitadas para escolher ao fazer suas reclamações.

[3] https://guarapuava.atende.net/autoatendimento/servicos/ouvidoria-municipal

3 SISTEMA ÁGORA

Neste capítulo serão apresentadas as funcionalidades do sistema (seção 3.1) e como o uso destas auxilia o usuário no domínio (seção 3.2).

3.1 Escopo

O sistema Ágora é uma iniciativa para favorecer e apoiar a fiscalização cidadã dos espaços públicos. Visa fornecer ao cidadão um canal onde pode relatar a necessidade de melhorias em espaços públicos, como, por exemplo, o conserto de vias e sua limpeza. As solicitações podem ser acessadas sem restrição e, dessa forma, não é necessário ter uma conta ou estar autenticado no sistema para ter acesso às solicitações já registradas. Entretanto, o cadastro e a edição de uma solicitação precisam da autenticação de um usuário.

O sistema tem como princípio prover uma forma prática e eficiente para registrar e acompanhar relatos de problemas urbanos de diversos tipos. Para tanto, foi projetado para funcionar no celular do usuário. Com isso, quando uma situação o incomodar, o usuário pode reforçá-la: desta forma, ela terá um impacto maior na cidade. O sistema não é uma ferramenta para a procura de problemas na cidade, mas para que cada pessoa possa, durante seus deslocamentos de trabalho, estudos ou lazer, relatar um problema com o qual se deparou na sua rotina. Este comportamento se torna massivo à medida que mais pessoas irão se tornar usuários do sistema.

O sistema é um produto voltado para o público em geral poder ajudar no cuidado de sua cidade. Contudo, pode ser usado por entidades públicas ou privadas, como forma de mapeamento de problemas urbanos dentro de uma determinada localidade, pois, desta forma, poderá ser possível entender quais os problemas mais comuns enfrentados pelos cidadãos e quais bairros ou locais mais afetados por essas situações.

Visando o tratamento das necessidades identificadas, o sistema apresenta as seguintes funcionalidades em relação às solicitações de manutenção urbana: (a) Cadastro; (b) Acompanhamento; (c) Reforço e priorização; e (d) Mapeamento.

3.2 Perspectivas

Os cidadãos, de todas as cidades, enfrentam problemas urbanos diariamente, e a ausência de um canal centralizado e especializado para informar essas situações acaba deixando o cidadão sem saber a quem recorrer. O sistema Ágora, proposto neste documento, proverá as funcionalidades necessárias para que o cidadão possa ser um agente fiscalizador, relatando situações inconvenientes encontradas na sua cidade ou em outras cidades por ocasião de uma viagem.

Com o uso do sistema Ágora, no seu celular, o usuário pode registrar solicitações de manutenção para problemas urbanos. Durante o cadastro da sua solicitação, o usuário pode escolher a categoria mais adequada ao tipo da ocorrência e adicionar fotos para complementar o cadastro. Após isso, com o cadastro da solicitação, ela é adicionada ao mapa usando a localização do dispositivo via Sistema de Posicionamento Global, do inglês Global Positioning System (GPS), de forma que outros usuários que estejam em uma área de 300 metros quadrados ao redor possam ver onde fica localizada a situação relatada pela solicitação do usuário. Quando uma solicitação é atendida, o usuário poderá finalizá-la.

Em alguns casos, pode acontecer de uma solicitação já estar registrada no sistema. Nesses casos, o usuário não precisa criar uma nova solicitação; ele pode simplesmente deixar seu apoio para a solicitação existente. Desta forma, a solicitação terá um peso maior através de um ranqueamento, no qual solicitações com maior apoio tendem a ser problemas que causam transtornos para uma maior quantidade de pessoas e, por isso, devem ser consideradas prioritárias.

Para o acompanhamento das solicitações, o usuário, ao criar uma solicitação, pode, em um processo de atualização de etapa, efetuar seu fechamento. Se a situação foi resolvida ou está em fase de obras, por exemplo, o usuário pode atualizar sua situação atual e enviar uma imagem como evidência dessa mudança. Além disso, as solicitações possuem categorias diferentes, permitindo que diversos tipos de problemas urbanos sejam relatados, pois qualquer tipo de problema merece atenção.

O sistema será uma grande vitrine da situação da infraestrutura pública de uma cidade, acessível por todo e qualquer cidadão, inclusive por aqueles que exercem cargo público. Dessa forma, os problemas existentes e relatados serão de conhecimento de todos. Da mesma forma, as benfeitorias e consertos realizados.

4 MATERIAIS E MÉTODOS

Neste capítulo, serão apresentadas as tecnologias, ferramentas e a metodologia utilizadas no desenvolvimento do sistema Ágora. Na seção 4.1, são descritas as ferramentas, bibliotecas, frameworks e plataformas de desenvolvimento utilizadas. Por fim, na seção 4.2, são detalhadas as atividades e tarefas de engenharia de software envolvidas no desenvolvimento do sistema.

4.1 Materiais

Sobre as tecnologias utilizadas, é possível separá-las em três categorias, que são as tecnologias do *backend* (API), *frontend* (aplicativo móvel) e outras ferramentas de uso geral.

O desenvolvimento do *backend* do sistema Ágora é realizado por meio da implementação de uma API, utilizando o framework Hypertext Preprocessor (PHP) *Laravel*[1], que se trata de um framework escalável e progressivo, que permite o desenvolvimento tanto de aplicações monolíticas quanto de APIs, contando com uma documentação abrangente e uma comunidade ativa (LARAVEL, 2023a). Para o armazenamento de dados, o sistema utiliza o PostgreSQL [2], um sistema de gerenciamento de banco de dados relacional robusto e amplamente adotado (POSTGRESQL, 2023).

O ambiente de desenvolvimento da API é configurado utilizando o Docker, uma ferramenta que permite implantar aplicações em contêineres, trazendo diversas vantagens para o desenvolvimento e deploy (DOCKERINC, 2023a). Para aproveitar ao máximo os benefícios do Docker, é utilizado em conjunto o Docker Compose, que possibilita a execução de múltiplos contêineres simultaneamente. A configuração desses contêineres é definida em um arquivo com extensão YML, uma abreviação de YAML Ain't Markup Language (YAML), onde são especificadas as imagens, os contêineres e suas respectivas configurações (DOCKERINC, 2023b).

Para realizar os testes *backend*, foram utilizadas as próprias ferramentas do *Laravel*, que, por si só, oferecem um amplo suporte para testes, trabalhando em conjunto com o PHPUnit [3]. Esses testes podem ser unitários, que verificam partes pequenas e isoladas do código, ou de funcionalidades, que, por sua vez, avaliam o funcionamento integrado dessas partes. Os testes de funcionalidades, geralmente os mais utilizados em uma aplicação, garantem maior segurança ao código (LARAVEL, 2023b).

Além da API, também foi desenvolvido um aplicativo *mobile* utilizando o framework React Native [4]. Essa tecnologia, baseada na biblioteca React [5], permite a criação de aplicativos nativos para Android e IOS utilizando código-fonte em JavaScript (META, 2023). Para facilitar o acesso

[1] https://laravel.com/
[2] https://www.postgresql.org/
[3] https://phpunit.de/index.html
[4] https://reactnative.dev/
[5] https://react.dev/

a recursos nativos dos dispositivos, o desenvolvimento do aplicativo também contou com o uso do Expo [6], uma plataforma que oferece um conjunto de ferramentas para simplificar o processo de criação e teste de aplicativos.

Além das tecnologias de desenvolvimento, foram utilizadas ferramentas de versionamento de código-fonte, como o Git e o GitHub. O Git é um sistema de controle de versão open-source gratuito, cuja principal vantagem é a possibilidade de trabalhar com ramificações (branches) dentro do mesmo projeto.

Já o GitHub é uma plataforma de hospedagem de código em nuvem que permite acompanhar o projeto, gerenciar branches e utilizar diversas ferramentas, como testes e deploys automáticos (LONGEN, 2023). Além disso, será utilizada a funcionalidade GitHub Projects, que, segundo a documentação oficial, permite criar quadros no formato Kanban para organizar tarefas e otimizar o fluxo de trabalho diretamente no repositório do projeto (GITHUB, 2023).

Para a prototipação das interfaces do sistema, foi utilizada a plataforma Figma[7], que oferece recursos simples e avançados para a criação de protótipos. Segundo Santana (2023), o Figma apresenta diversas vantagens para o design de interfaces, pois é uma ferramenta online multiplataforma, acessível diretamente pelo navegador em qualquer computador com conexão à internet. Além disso, permite a colaboração em tempo real entre membros da equipe e dispõe de ferramentas que auxiliam na criação de layouts responsivos (SANTANA, 2023).

Para a modelagem do banco de dados, foi utilizado o Miro[8], com esta ferramenta foi desenvolvido o MER do sistema Ágora. O MER poderá ser atualizado durante a etapa de desenvolvimento, para ir se adaptando a novas demandas ou melhorias que podem ser feitas durante este processo. O uso da ferramenta Miro, segundo a sua documentação, tem como vantagens, além da versão para uso gratuito, funcionalidades que possibilitam a elaboração de um MER de forma fácil, pois têm diversos modelos pré-definidos (MIRO, 2023).

O processo de documentar o sistema é uma ótima prática durante o processo de desenvolvimento de software, porém nem sempre é uma prática adotada. Segundo um artigo disponibilizado pela Atlassian (2023a), elaborar a documentação pode trazer vantagens como, por exemplo, economia de tempo na procura de uma determinada funcionalidade ou, ainda, entender algum fluxo. Outra vantagem é poder ter maior controle na qualidade do seu software, evitar duplicação de tarefas, facilitar a integração de novos desenvolvedores e aumentar o nível de conhecimento coletivo da equipe, pois através da documentação, novas informações podem ser partilhadas de forma que todos possam ter acesso a elas (ATLASSIAN, 2023a).

Para documentar o desenvolvimento do sistema Ágora, foi utilizado um framework chamado Swagger[9], configurado na aplicação com o uso de um pacote disponível para o *Laravel*[10]. A sua escolha se deve ao fato de que, além de permitir documentar diretamente no código da

[6] https://expo.dev/
[7] https://www.figma.com
[8] https://miro.com
[9] https://swagger.io/
[10] https://github.com/DarkaOnLine/L5-Swagger

aplicação, oferece uma interface interativa acessível através de uma rota específica no navegador, construída a partir de um JSON que contém todo o conteúdo referente à documentação gerada. Isso permite testar os endpoints da API por meio da funcionalidade (*"Try it out"*), que possibilita a execução direta de requisições aos endpoints documentados. Dessa forma, é possível autenticar-se como usuário, testar diferentes chamadas e aplicar filtros, entre outras ações.

Além disso, o Swagger exibe os payloads esperados e as respostas retornadas, facilitando a análise das requisições. Também permite adicionar informações gerais sobre o software documentado, proporcionando uma experiência mais completa e prática para desenvolvedores e usuários.

4.2 Métodos

O desenvolvimento do sistema está organizado em ciclos. A metodologia de desenvolvimento elaborada para conduzir este projeto, para cada ciclo, como ilustrada na Figura 3, será composta das seguintes atividades: levantamento de requisitos, prototipação, testes e codificação.

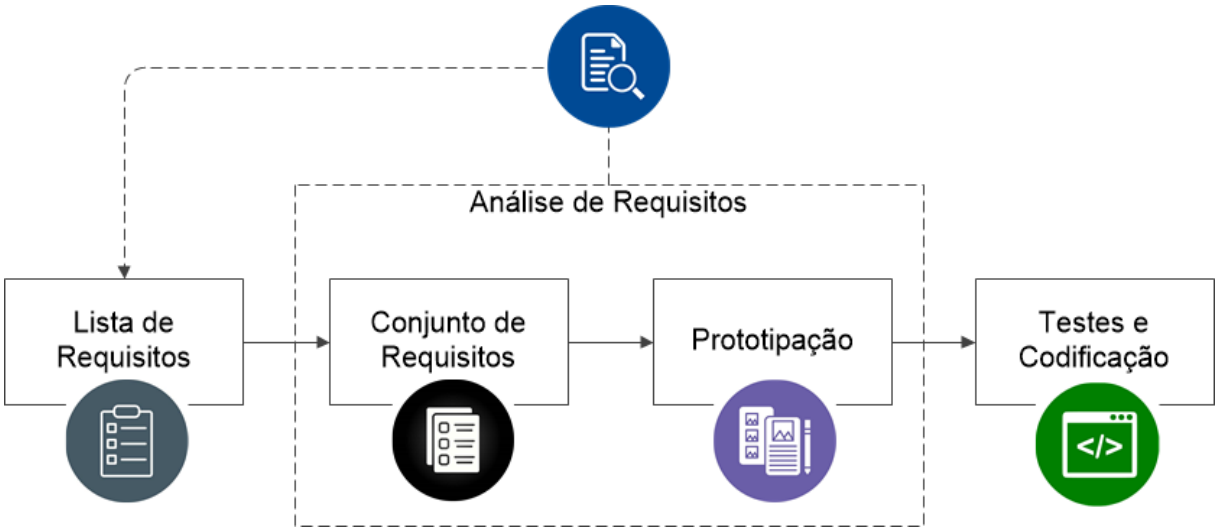


Figura 3 – Metodologia de desenvolvimento
Fonte: Autoria própria.

Uma parte essencial no desenvolvimento de software é a engenharia de requisitos. “Os requisitos de um sistema são as descrições do que o sistema deve fazer, os serviços que oferece e as restrições a seu funcionamento” (SOMMERVILLE, 2011a). Assim sendo, primeiramente será feito o levantamento dos requisitos mais tangíveis (**Lista de Requisitos**), que serão necessários para o sistema (SOMMERVILLE, 2011b).

Neste trabalho, a Lista de Requisitos é definida pela equipe de desenvolvimento, contendo somente requisitos funcionais. Contudo, à medida que o sistema se torne acessível para os usuários, será possível verificar e validar se os requisitos propostos atendem ao que o usuário espera da aplicação.A partir desta lista preliminar de requisitos, é possível quantificar o número

de tarefas e o tempo de desenvolvimento. Também é igualmente importante como auxílio para o processo de modelagem do banco de dados. A implementação do sistema tem início com as atividades de verificação e elaboração de requisitos (**Conjunto de Requisitos**), onde, por meio de reuniões, do entendimento do domínio, abrangência da aplicação e uso, um conjunto de requisitos da Lista de Requisitos será trabalhado, de forma a entender as funcionalidades e o que seria necessário ser desenvolvido. Após estas atividades, é iniciada a etapa de prototipação das telas para o sistema (**Prototipação**). Esse método permite testar/validar ideias antes da implementação e, também, descobrir falhas ou requisitos que não foram considerados (JUHL; SøRENSEN, 2023).

Por fim, considerando o apresentado na Figura 3, a implementação do *backend* do sistema tem continuidade por meio da codificação de testes e funcionalidades (**Testes e Codificação**). Para esta etapa de desenvolvimento do *backend*, é adotada a metodologia Test Driven Development (TDD), segundo Steinfeld (2024), esta abordagem, estabelece que o código deve ser escrito com base em testes unitários. Dessa forma, o código-fonte é desenvolvido a partir de um teste que falhou previamente e, em seguida, aprimorado até que o teste seja executado com sucesso. Assim, o teste define o comportamento esperado do sistema (STEINFELD, 2024). O uso desta metodologia traz grandes vantagens ao projeto, pois permite a criação de um código robusto e flexível, menos suscetível a falhas e a problemas decorrentes de refatorações ou alterações. Outro conceito fundamental no TDD é que o código deve ser fácil de testar (ACOSTA; GAJDA, 2023). Dessa maneira, a codificação do *backend* é iniciada com o desenvolvimento de testes unitários, utilizando as técnicas de TDD. Após a elaboração desses testes, o código é construído para garantir que todos sejam executados com sucesso. Por outro lado, as atividades do *frontend* são iniciadas diretamente com o desenvolvimento das funcionalidades, sem a criação prévia de testes.

Além do TDD, no *backend* foi optado por utilizar uma abordagem baseada no *Domain-Driven Design* (DDD), que, de acordo com Duarte (2018) é um padrão de design de software criado em 2003 por Eric Evans, publicado em um livro com o mesmo título. Esse padrão foca no domínio do problema, buscando compreendê-lo e solucioná-lo de maneira organizada (DUARTE, 2018). Partindo disto, foi adotado o padrão de separar dentro da pasta *app*,com os domínios presentes no sistema. Nesta abordagem, cada domínio pode ter regras de validação, Middlewares, Modelos, Jobs (processos executados de forma assíncrona), DTOs, que serão utilizados para transferir informações entre os controladores e as classes que farão o processamento, e, por último, mas não menos importante, as Actions, que se referem a classes que executam uma determinada função, baseadas no conceito de *use cases* (casos de uso). Na padronização adotada, cada Action possui uma função chamada 'execute', que deve fazer somente uma única ação. Desta forma, é possível testar individualmente as Actions mais importantes por meio de testes unitários. Além dessa vantagem, essas classes podem ser reutilizadas, tornando o código mais modular e eficiente.

A gestão dos requisitos acontece principalmente durante as atividades de verificação, elaboração e prototipação (**Análise de Requisito**), mas também durante a etapa de implementação do sistema (**Testes e Codificação**), fomentando o amadurecimento e evolução dos requisitos para o sistema. Consequentemente, alguns requisitos são atualizados ou substituídos ao longo do andamento do projeto. Entende-se que essa abordagem é válida, pois proporciona uma definição clara e precisa dos requisitos e, consequentemente, contribui para a evolução do sistema.

O desenvolvimento do sistema, suas funcionalidades e testes, compreendidos na etapa de **Testes e Codificação**, é conduzido utilizando o Git e o GitHub, como mencionado anteriormente. Será utilizado o fluxo de trabalho conhecido como Gitflow, descrito mais adiante nesta seção. Neste momento, a criação de código será organizada conforme apresentado na Figura 4, seguindo um microfluxo para o gerenciamento e controle desta etapa, no qual estão presentes os conceitos de funcionalidade (feature) e desenvolvimento (develop). Esses conceitos são:

- Feature: De acordo com o Atlassian (2023b), as ramificações de feature, tem como pai a ramificação develop, cada uma delas, vai conter o desenvolvimento de uma ou um conjunto de funcionalidades inerentes, que quando é concluído é realizado merge[11] para a develop (ATLASSIAN, 2023b). Desta forma, o código da develop fica atualizado com as novas funcionalidades implementadas na ramificação de desenvolvimento;
- Develop: É a ramificação de desenvolvimento dentro do fluxo do Gitflow. Segundo Atlassian (2023b), está ramificação é alimentada pelas features, quando já possui recursos suficientes para uma nova versão do software, é realizado uma bifurcação a partir desta ramificação para uma branch release, a qual conterá o conteúdo na nova versão disponibilizada (ATLASSIAN, 2023b).

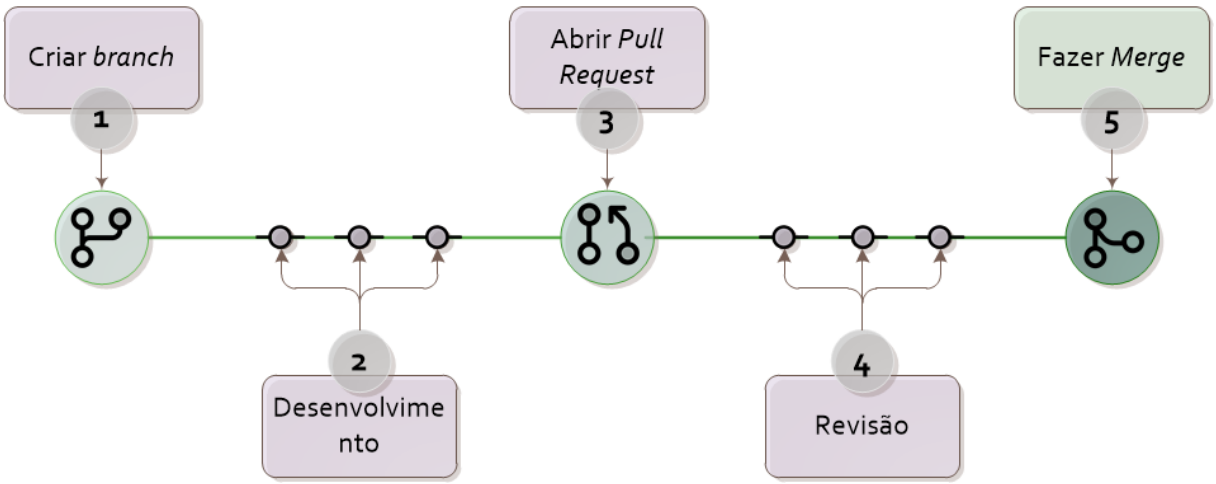


Figura 4 – Fluxo de desenvolvimento.
Fonte: Autoria própria.

[11] Processo de mesclar duas diferentes ramificações

Após a elaboração do desenvolvimento da atividade e da validação com testes unitários, a próxima etapa era referente aos testes de funcionalidade para a API (2º passo). O objetivo deste tipo de teste é poder validar as funcionalidades de forma completa. Com os testes implementados, e o código correspondendo de acordo com o desejado, é feita uma validação geral da atividade, para verificar se:

- (a) O requisito foi realmente cumprido;
- (b) Essa atividade pode gerar impacto em alguma outra área ou regra do sistema;
- (c) O código está escrito de acordo com boas práticas utilizando a ferramenta Pint (LARAVEL, 2023c), esta, integrada com a plataforma GitHub, para que todo o código novo seja verificado.

Por fim, em todo ciclo de desenvolvimento é gerado uma Pull Request (3º passo), uma solicitação para realizar o merge do novo código no projeto. Posteriormente, no 4º passo, este código é revisado pelo professor orientador para verificar a codificação, a arquitetura e o desempenho da aplicação, inclusive, realizando a execução dos testes unitários e de funcionalidade novamente, bem como fazendo um teste de uso do sistema. Após a conclusão, o atual ciclo de desenvolvimento pode ser considerado completado e o novo código pode ser agrupado ao sistema, no 5º passo da Figura 4.

O fluxo de desenvolvimento de cada ciclo não é uma estrutura rígida, se em algum ponto durante a atividade de desenvolvimento, for notado algum erro, inconsistência ou má descrição, pode-se retornar para a atividade anterior (Análise de Requisitos), a fim de que a atividade possa ser desenvolvida da melhor forma possível.

O micro fluxo da Figura 4 está inserido dentro de um fluxo maior, o Gitflow, representado na Figura 5, mais precisamente como feature.

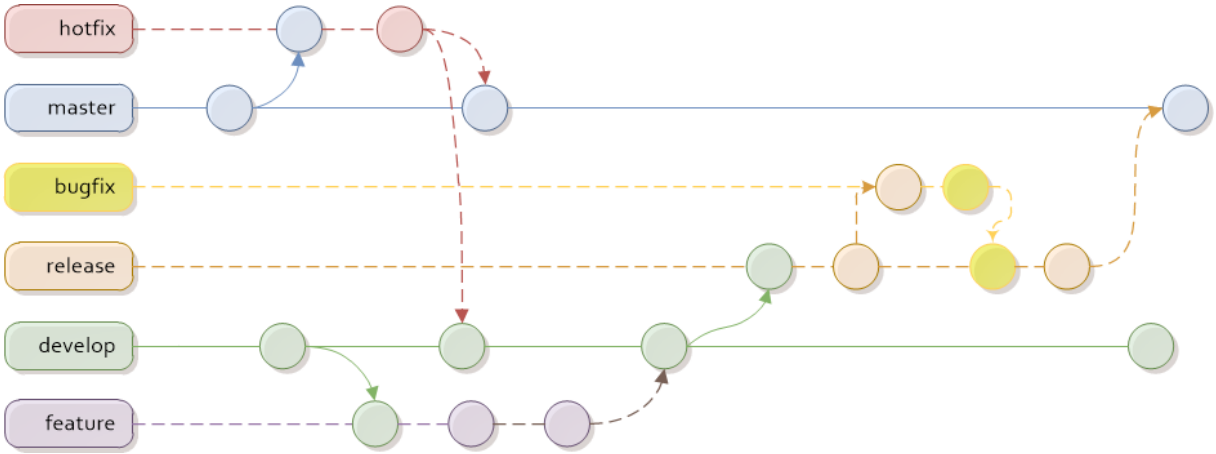


Figura 5 – Fluxo de Ramificações
Fonte: Autoria própria.

Como foi previamente mencionado, foi criada uma nova ramificação a partir da branch develop. De acordo com Atlassian (2023b), esta nova branch é uma release. Assim, quando

esta nova versão estiver pronta para ser lançada, deve ser feito o merge dessa ramificação com a branch master, a principal branch do repositório, representando a versão mais estável do software. Caso sejam encontrados erros de código, para realizar as correções necessárias, devem ser utilizadas ramificações específicas para esse propósito. Se o erro for encontrado na branch develop, deve ser criada uma ramificação do tipo bugfix. Após a correção, o merge é realizado na branch develop. Por outro lado, se o erro for encontrado na versão estável do software, ou seja, na branch master, é criada uma ramificação do tipo hotfix. Essas branches são elaboradas para correção de erros que ocorrem na versão disponibilizada do software, que pode estar, muitas vezes, em ambiente de produção. Em um cenário onde coexistem uma branch hotfix e uma release, a correção deve ser aplicada também na branch release (ATLASSIAN, 2023b).

5 ANÁLISE DO SISTEMA

Antes do início do desenvolvimento do sistema Ágora, foi realizada uma análise inicial com base na metodologia descrita na seção 4.2. Essa etapa teve como objetivo definir as expectativas e funcionalidades do sistema, garantindo um direcionamento sólido para sua implementação. Como resultado dessa análise, foram elaborados dois itens principais: a lista de requisitos e o MER. Nesta seção, esses elementos serão apresentados e detalhados, respectivamente, nas subseções seção 5.1 e seção 5.2.

5.1 Lista de Requisitos

Após a análise das funcionalidades esperadas do sistema, foram criados, neste primeiro momento, 16 requisitos funcionais. Na Tabela 1 estão todos esses requisitos, cada item na tabela contém colunas para exibir seu número identificador, título e descrição.

Tabela 1 – Lista de Requisitos do Sistema

Número	Título	Descrição
RF 001	Cadastro de novos usuários.	Como futuro usuário do sistema, quero poder realizar meu cadastro, para ter acesso ao aplicativo.
RF 002	Edição de Usuários	Como usuário, quero poder atualizar as minhas informações pessoais e de acesso, para manter a integridade das minhas informações.
RF 003	Login de usuário.	Como usuário, quero poder utilizar minhas credenciais, para realizar o login.
RF 004	Logout de usuário.	Como usuário, quero poder encerrar a minha sessão, para evitar acessos indevidos em meu nome.
RF 005	Cadastro de solicitações de manutenção urbana.	Como usuário, quero poder registrar solicitações de manutenção urbana, para tornar público um problema ou demanda na cidade.
RF 006	Utilizar a localização atual no momento de cadastrar uma solicitação de manutenção urbana.	Como usuário, quero que minha localização seja considerada no momento do registro de uma solicitação de manutenção urbana, para que o problema ou demanda sejam encontrados com maior facilidade.
RF 007	Adição de imagens ao realizar o cadastro de uma solicitação de manutenção urbana.	Como usuário, quero adicionar fotos a uma solicitação de manutenção urbana, para mostrar ou validar a sua situação atual do problema ou demanda.
RF 008	Categorizar as solicitações de manutenção urbanas.	Como usuário, quero categorizar a solicitação de manutenção urbana, para organização do problema ou demanda.
RF 009	Edição de solicitações de manutenção urbanas.	Como usuário, eu quero poder editar as informações das solicitações de manutenção urbana criadas por mim.
RF 010	Exclusão de solicitações de manutenção urbanas.	Como usuário, eu quero poder excluir solicitações de manutenção urbana criadas por mim.

Continua na próxima página

Tabela 1 – Continuação da Página Anterior

Número	Título	Descrição
RF 011	Exibição de solicitações de manutenção urbanas no mapa.	Como usuário, eu quero poder visualizar solicitações de manutenção urbana criadas por mim e por outros usuários.
RF 012	Exibição de uma solicitação de manutenção urbana em particular.	Como usuário, eu quero poder visualizar uma solicitação de manutenção urbana, podendo ver seus detalhes, como por exemplo, imagens e histórico.
RF 013	Poder solicitar o fechamento de uma solicitação de manutenção urbana.	Como usuário, eu quero poder realizar o fechamento de uma solicitação de manutenção urbana criada por mim.
RF 014	Funcionalidade de reforço a solicitações de manutenção urbana já registradas.	Como usuário, quero poder reforçar determinada solicitação, para que esse problema ganhe um grau maior de visibilidade e poder demonstrar que sou afetado pela mesma ocorrência.
RF 015	Aplicação de filtros para exibição de registros no mapa.	Como usuário, quero poder aplicar filtros nas listagens, para poder ver somente solicitações de manutenção urbana específicas de acordo com os critérios dos filtros.
RF 016	Casos possíveis para deletar uma solicitação de manutenção urbana	Como usuário, quero poder excluir minhas solicitações de manutenção urbanas, somente caso ela não ter sido alterada por outro usuário.

5.2 Modelo Entidade Relacionamento (MER)

Inicialmente, o MER do sistema Ágora, representado na Figura 6, foi elaborado com as principais entidades do sistema, independentemente da tecnologia. Esse modelo serviu para a definição das entidades e dados. No momento de desenvolvimento do banco de dados em um Sistema de Gerenciamento de Banco de Dados (SGBD), a implementação seguiu as particularidades e regras do banco de dados escolhido, além das especificidades do framework *Laravel*, para garantir compatibilidade com o *Object-Relational Mapping* (ORM) do framework, o Eloquent [1].

No MER desenvolvido, a tabela chamada *users* é responsável pelo armazenamento das informações referentes à autenticação dos usuários. Assim, todo novo usuário da plataforma precisa ter um registro nesta tabela. Campos contendo informações sensíveis, como o campo de senha, são registrados de forma criptografada, como medida de segurança. Esta tabela tem relacionamento com a tabela *addresses*, que armazena informações sobre o endereço atual dos usuários. Essas informações são usadas exclusivamente para relatórios gerais sobre o uso do sistema. Cada usuário deve ter apenas um endereço cadastrado.

A principal entidade no MER desenvolvido corresponde à tabela *solicitations*, onde são registradas as solicitações de manutenção urbana, incluindo título e descrição. Além disso, há campos responsáveis por controlar a exibição das solicitações no mapa e o fluxo de processos. Para isso, o campo *status* indica a situação de cada solicitação (por exemplo, ativa ou resolvida).

[1] https://laravel.com/docs/10.x/eloquent

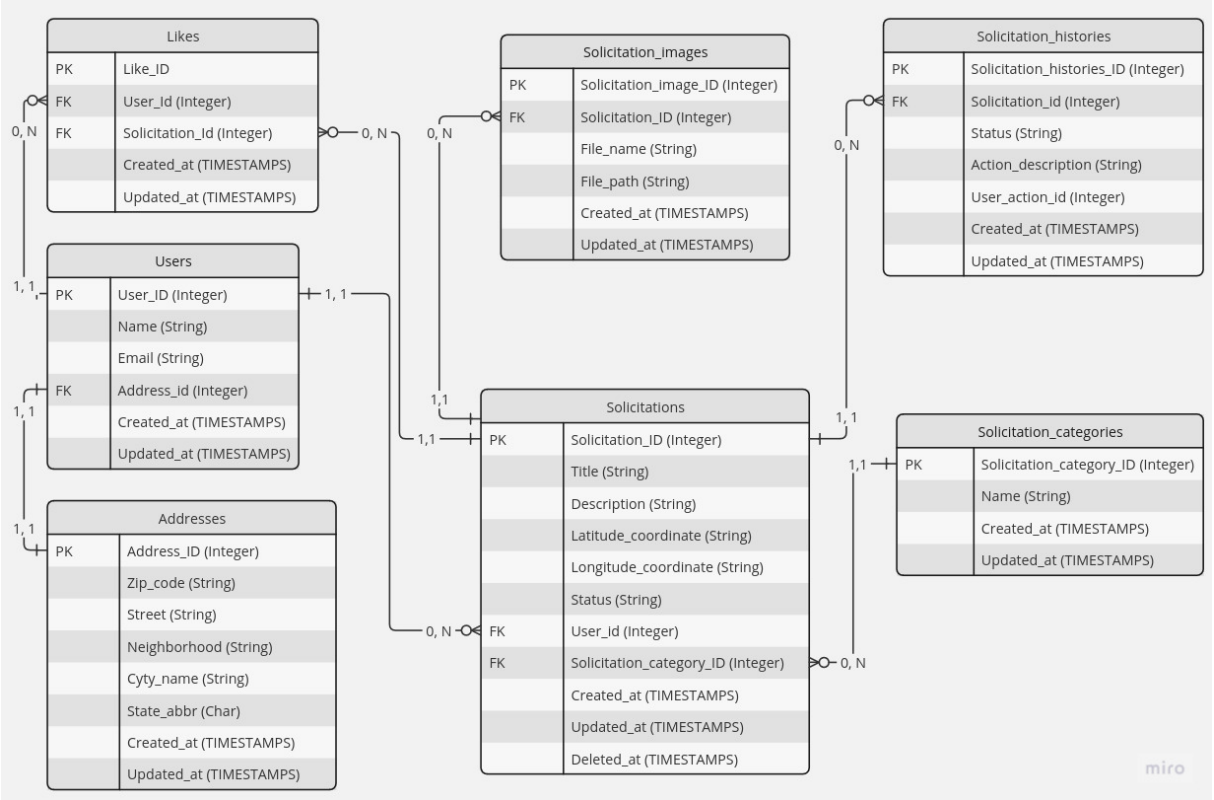


Figura 6 – Diagrama MER do sistema
Fonte: Autoria própria.

Para gerenciar os valores possíveis desse campo, pode ser utilizado um Enum, uma classe que define um conjunto de valores permitidos para um tipo específico. Os valores do Enum para as possíveis situações foram definidos em uma etapa futura do projeto. Além disso, essa tabela se relaciona com a tabela *solicitation_categories*, que é usada para classificar os diferentes tipos de solicitações registradas.

Cada ação realizada em uma solicitação de manutenção urbana resulta em um novo registro na tabela *solicitation_histories*', permitindo rastrear todas as ações tomadas em relação a uma solicitação. Por exemplo, quando um usuário sugerisse o fechamento de uma solicitação, essa ação seria registrada, incluindo a situação da solicitação naquele momento, a descrição da ação realizada e o identificador do usuário responsável. As solicitações poderiam ter múltiplos registros desse tipo em seu histórico simultaneamente.

Como parte essencial no processo de registro de novas solicitações de manutenção urbana, o MER conta com uma entidade responsável por armazenar e relacionar imagens a uma solicitação. No diagrama, essa entidade corresponderia à tabela *solicitation_images*.

Por fim, a tabela likes é responsável por registrar os reforços que as solicitações podem receber dos usuários do sistema. Cada usuário pode registrar seu reforço apenas uma vez para cada solicitação.

6 AMBIENTE DE DESENVOLVIMENTO

Para a elaboração do ambiente de desenvolvimento do projeto, foi necessário seguir algumas etapas, como configuração do framework *Laravel* para ser executado como API, integração com banco de dados e criação de um projeto Expo com React Native.

Prezando pela melhor organização do versionamento do *frontend* e *backend* do projeto, foram criados repositórios separados no GitHub como plataforma de versionamento. Cada um desses repositórios conta com uma descrição detalhada do processo de instalação e execução após serem clonados a partir do GitHub.

Após a configuração inicial dos repositórios, foi necessário criar um projeto *Laravel*, utilizado como API do projeto Ágora. Esse processo foi realizado seguindo as recomendações da documentação do *Laravel*, que indica a instalação do Composer de forma global na máquina e, em seguida, a criação de um novo projeto utilizando o comando adequado (LARAVEL, 2024a). A partir disso, com o projeto já criado, foram adicionados os arquivos Docker de forma que o projeto inicial fosse integrado às configurações do ambiente de desenvolvimento.

Para isso, dois conceitos são importantes: volumes e imagens. Volumes são mecanismos usados para persistir dados que são gerados e usados por contêineres Docker. Seu uso traz vantagens, como a possibilidade de compartilhar o mesmo volume com um ou mais contêineres. Além disso, são uma opção melhor para persistir dados, pois, além de não aumentarem o tamanho do contêiner, ficam fora do seu ciclo de vida (DOCKERINC, 2024a). E as imagens, no ambiente Docker, correspondem a pacotes elaborados que podem conter arquivos, binários, bibliotecas e configurações (DOCKERINC, 2024b). O uso dessas imagens pré-definidas traz grandes vantagens ao desenvolvimento e à implantação de aplicações, pois muitas delas já incluem configurações complexas. O Docker disponibiliza, no site Docker Hub [1], várias imagens, que podem ser distribuições oficiais de serviços ou até mesmo elaboradas pela comunidade.

A configuração dos contêineres Docker foi elaborada em um arquivo com a extensão YML, chamado *docker-compose.yml*, representado na Figura 7, com as seguintes configurações:

a) Versão que será executado o Docker Compose;

b) Quais os volumes que serão armazenados no ambiente de desenvolvimento;

c) As imagens utilizadas nos contêineres da aplicação e configurações necessárias.

Este arquivo coordena o ambiente de desenvolvimento. De acordo com suas configurações, ele cria os seguintes componentes:

a) A versão selecionada para o Docker Compose;

[1] https://docs.docker.com/guides/docker-concepts/the-basics/what-is-an-image/

b) O volume que será utilizado, chamado *agora-app-data*;

c) Os serviços e seus nomes: o *agora-app* utiliza uma imagem com PHP na versão 8.2, juntamente com uma imagem Alpine para executar a aplicação. Para o banco de dados foi utilizada uma imagem do PostgreSQL e Alpine, com as devidas configurações de acesso ao banco.

Outro arquivo vital está na pasta *environments* do diretório raiz do projeto. Este arquivo, nomeado *setup.sh*, é demonstrado na Figura 8. Nele estão contidos os comandos que são executados automaticamente para a configuração inicial do projeto, como copiar o conteúdo do arquivo *.env-example* para o arquivo *.env*, instalar as dependências do projeto, gerar a sua chave única e executar as Seeders iniciais da aplicação. As Seeders, de acordo com a documentação do *Laravel*, são basicamente arquivos que populam o banco de dados com informações pré-definidas nas classes (LARAVEL, 2024b).

Para a reestruturação e padronização da API, foi necessário realizar mudanças na estrutura de diretórios do projeto usando o framework *Laravel*. Pois, de maneira análoga a diversos frameworks, o *Laravel* disponibiliza uma estrutura padrão de diretórios [2]. Dentre esses diretórios, destaca-se a pasta *app*, que, conforme a documentação do *Laravel*, contém o código principal da aplicação (LARAVEL, 2024c). Por padrão, esse diretório abriga subdiretórios como *Console* e *Http*. Este último, segundo a documentação do *Laravel*, reúne os principais arquivos responsáveis pelo controle de entrada e saída de dados, incluindo os controladores [3], Requests [4], Middlewares [5], Resources [6] e o arquivo *Kernel*, no qual estão registrados os Middlewares utilizados na aplicação. Tendo como base esta estrutura inicial, foram feitas refatorações aplicando a metodologia DDD, que resultou em uma estrutura de diretório mais organizada; esta estrutura está sendo exibida na Figura 9 junto com um diagrama exemplificando a organização utilizada.

Além das mudanças nos diretórios mencionadas, foram adicionados arquivos base voltados para a elaboração e execução de testes. Esses arquivos tiveram como principal objetivo padronizar a execução dos testes, garantindo que testes unitários não interagissem com o banco de dados, além de agrupar funções reutilizáveis em cenários de testes de feature. Eles também permitem a execução de classes destinadas a popular o banco de dados e preparar cenários específicos para os testes. Entre esses arquivos, chamados de cases (casos), destaca-se o arquivo de casos para testes unitários, exemplificado na Figura 10. Esse arquivo foi projetado para impedir qualquer interação da classe testada com o banco de dados; caso tal interação ocorra, o teste é automaticamente marcado como falho.

Outra medida adotada foi a instalação da biblioteca Pint, juntamente com a criação de um comando para sua execução. Como se trata de um pacote oficial do *Laravel*, sua instalação

[2] https://laravel.com/docs/11.x/structure
[3] https://laravel.com/docs/11.x/controllers
[4] https://laravel.com/docs/11.x/requests
[5] https://laravel.com/docs/11.x/middleware
[6] https://laravel.com/docs/11.x/eloquent-resources

é simples e detalhadamente descrita na documentação oficial [7]. Após a instalação, foi criado um arquivo de configuração no diretório raiz da aplicação, conforme ilustrado na Figura 11. Esse arquivo permite a personalização das regras de estilização do código. No contexto do projeto, foi configurado para excluir as pastas de Migrations e Seeders da verificação de estilização, preservando o padrão gerado pelo *Laravel*.

Por fim, foi adicionado um comando no arquivo *composer.json*, como exibido na Figura 12, para facilitar a execução da verificação e correção de estilização do código PHP.

Como mencionado anteriormente, na seção 4.1, a ferramenta escolhida para documentar a API foi o Swagger. Assim como o sistema, foi desenvolvida de forma incremental: a cada novo ciclo de desenvolvimento que introduzia funcionalidades e endpoints, eram adicionados os respectivos trechos documentando essas alterações. A documentação foi elaborada diretamente nos controladores, detalhando as respostas esperadas para cada endpoint e especificando claramente as informações necessárias para cada requisição, e a documentação gerada está disponível em https://github.com/GabrielPeplinski/agora-api/blob/main/project/storage/api-docs/api-docs.json.

Para garantir a qualidade do código antes de realizar o merge de novas alterações, foram criados workflows (fluxos de trabalho) específicos para validar alguns cenários. O GitHub permite configurar e executar workflows automatizados. De acordo com sua documentação, os workflows são descritos em arquivos no formato YML, que devem ser armazenados em um diretório específico na raiz do projeto, denominado *.github/workflows*. Cada fluxo de trabalho é composto por um ou mais eventos que atuam como gatilhos, ou seja, disparam sua execução, além de tarefas que especificam as ações a serem realizadas e os scripts associados para executar essas ações (GITHUB, 2025).

Esses workflows incluem: a verificação de todos os testes escritos na API, tanto os unitários como os funcionais; a validação da documentação gerada com o Swagger, assegurando que esteja de acordo com a sintaxe utilizada, seguindo padrões e formatação; e a análise da estilização do código PHP por meio da ferramenta Pint. Todos esses fluxos de trabalho foram configurados para serem executados automaticamente ao criar uma nova Pull Request.

Na Figura 13, é apresentado o *workflow* responsável por executar o Pint, verificando se o código segue os padrões estabelecidos pela aplicação. Caso todos os *workflows* sejam concluídos com sucesso, a Pull Request estará apta para ser mesclada ao código principal. Esse cenário é ilustrado na Figura 14.

Para criar um novo projeto Expo, referente ao aplicativo que irá consumir a API, foi necessário instalar o Node na máquina local. Após essa etapa, bastou seguir as orientações disponíveis na documentação oficial do Expo [8].

[7] https://laravel.com/docs/11.x/pint

[8] https://docs.expo.dev/get-started/create-a-project/

Com base no projeto padrão fornecido pelo Expo, não foram realizadas mudanças significativas na estrutura inicial. No entanto, foram instaladas algumas bibliotecas essenciais para a aplicação, entre elas:

1. React Native Paper [9] foi escolhida como a biblioteca principal de componentes devido à sua versatilidade e à ampla variedade de componentes prontos. Essa escolha trouxe diversos benefícios, como maior agilidade no desenvolvimento, padronização moderna e interfaces visuais agradáveis. Para integrá-la ao projeto, utilizou-se um *Context*[10], uma funcionalidade do React que permite compartilhar dados acessíveis em diferentes níveis da árvore de componentes. Isso é especialmente útil para informações globais, eliminando a necessidade de passar dados como *props* (propriedades) por múltiplos componentes. O uso de um *Context* é simples: basta garantir que os componentes que dependem dele estejam englobados pelo provedor do contexto. A Figura 15 ilustra um exemplo prático de como o React Native Paper foi integrado ao projeto por meio dessa funcionalidade.

2. Axios [11], adotado como cliente HTTP pela sua facilidade de uso e popularidade. Conforme destacado por Chaves (2023), o Axios oferece grande compatibilidade com navegadores e funcionalidades úteis, como interceptadores de resposta. Todas essas características tornam o Axios uma excelente opção para aplicações que consomem APIs(CHAVES, 2023). Como o Axios é uma biblioteca externa, foi criada uma instância em um arquivo separado para centralizar sua configuração, garantindo que todas as implementações e usos na aplicação seguissem as mesmas definições. Esse arquivo está localizado na pasta *utils*, no diretório principal, destinada a armazenar arquivos genéricos reutilizáveis em diferentes partes da aplicação. Este arquivo de configurações está sendo exibido na Figura 16.

3. Formik [12], utilizada como a biblioteca padrão para manipulação de formulários. Conforme descrito em sua documentação, essa ferramenta facilita e simplifica a implementação de formulários em aplicações React, fornecendo recursos otimizados para gerenciar estados e validações (FORMIK, 2025).

4. Yup [13], descrita em sua documentação como uma biblioteca que oferece diversas validações de formulários pré-definidas e personalizáveis, permitindo a adição de regras específicas e mensagens de erro customizadas conforme a necessidade (YUP, 2025). A documentação oficial do Formik recomenda o uso do Yup, devido à sua integração

[9] https://reactnativepaper.com/
[10] https://legacy.reactjs.org/docs/context.html
[11] https://axios-http.com/ptbr/docs/intro
[12] https://formik.org/docs/guides/react-native
[13] https://github.com/jquense/yup?tab=readme-ov-file

nativa, o que facilita a aplicação de validações de maneira eficiente e direta nos formulários.

```yaml
volumes:
  app-postgres-data:
    driver: local

services:
  app:
    user: 1000:1000
    container_name: agora-back-app
    image: webdevops/php-nginx:8.2-alpine
    volumes:
      - ./project:/app
    working_dir: /app
    environment:
      - WEB_DOCUMENT_ROOT=/app/public
      - PHP_MEMORY_LIMIT=256M
      - php.opcache.enable_cli=1
    ports:
      - 8080:80

  postgres:
    container_name: agora-back-postgres
    image: postgres:16-alpine
    volumes:
      - app-postgres-data:/var/lib/postgresql/data
    environment:
      - POSTGRES_DB=app_dev
      - POSTGRES_USER=postgres
      - POSTGRES_PASSWORD=laraveldocker
    ports:
      - 5433:5432

  postgres-test:
    container_name: agora-back-postgres-test
    image: postgres:16-alpine
    environment:
      - POSTGRES_DB=app_dev
      - POSTGRES_USER=postgres
      - POSTGRES_PASSWORD=laraveldocker
    ports:
      - 5434:5432
    tmpfs:
      - /var/lib/postgresql/data

  redis:
    container_name: agora-back-redis
    image: redis:5.0
    command: --appendonly no
```

Figura 7 – Arquivo *docker-compose.yml*, com as configurações de imagens e contêineres Docker para o ambiente de desenvolvimento do sistema Ágora.

Fonte: Autoria própria.

```bash
#!/bin/bash

docker compose exec app cp .env.example .env
docker compose exec app cp .env.testing.example .env.testing

docker compose exec app composer install
docker compose exec app php artisan migrate:fresh --seed
docker compose exec app php artisan key:generate
docker compose exec app php artisan storage:link
```

Figura 8 – Arquivo de comandos de instalação e configurações iniciais, utilizado para configurar o ambiente pela primeira vez.

Fonte: Autoria própria.

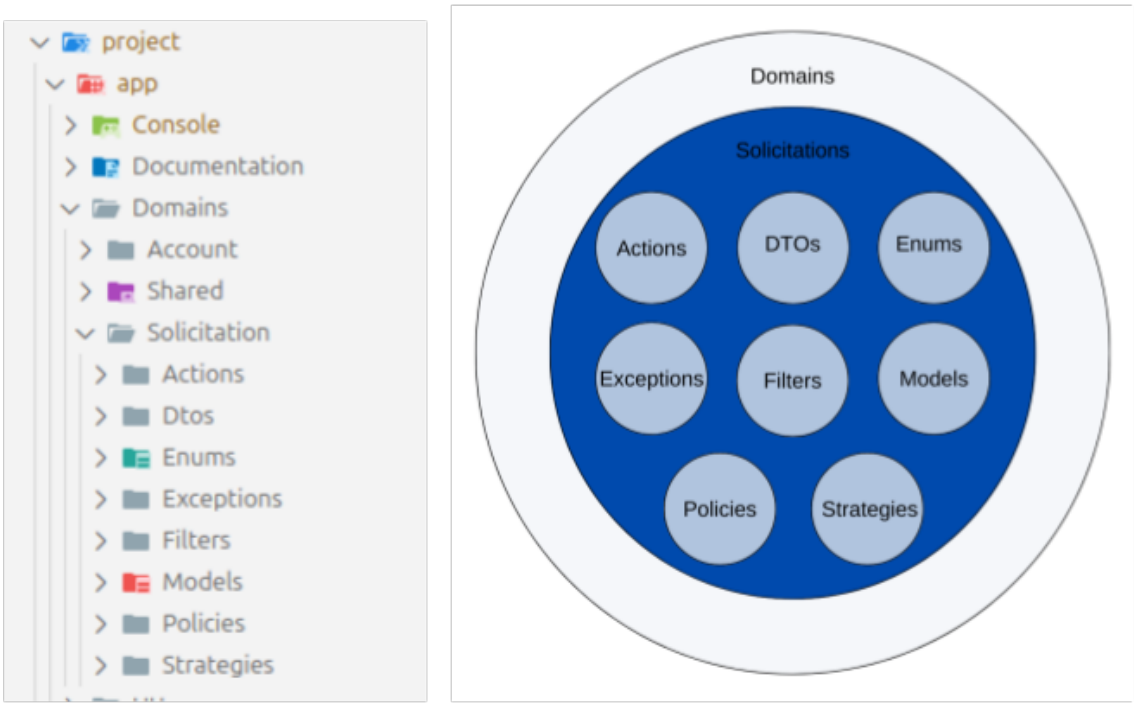


Figura 9 – Estrutura de diretórios baseada no DDD utilizada na API e diagrama para demonstrar de forma mais lúdica esta organização.

Fonte: Autoria própria.

```php
<?php

namespace Tests\Cases;

use Illuminate\Database\Events\StatementPrepared;
use Illuminate\Foundation\Testing\TestCase as BaseTestCase;
use Illuminate\Support\Facades\Event;
use Tests\CreatesApplication;

abstract class TestCaseUnit extends BaseTestCase
{
    use CreatesApplication;

    protected function setUp(): void
    {
        parent::setUp();
        $this->preventSqlQueries();
    }

    private function preventSqlQueries(): void
    {
        Event::listen(StatementPrepared::class, static function () {
            $message = 'SQL queries are not allowed in unit tests.';
            throw new \Exception($message);
        });
    }
}
```

Figura 10 – Arquivo de caso de testes unitários.
Fonte: Autoria própria.

```json
{
    "exclude": [
        "database/seeders",
        "database/migrations"
    ]
}
```

Figura 11 – Arquivo de configurações de estilização de código para a ferramenta Pint.
Fonte: Autoria própria.

```
"scripts": {
        "pint": [
            "./vendor/bin/pint"
        ],
    },
```

Figura 12 – Comando de execução de verificação e correção de estilização de código PHP utilizando Pint.

Fonte: Autoria própria.

```yaml
name: API Styleguide Check (Pint)

on:
  push:
    branches:
      - main
  pull_request:
    paths:
      - '**.php'
      - 'phpunit.xml'
      - '.github/workflows/style.yml'
      - 'composer.lock'

jobs:
  style:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - name: "laravel-pint"
        uses: aglipanci/laravel-pint-action@2.0.0
        with:
          preset: laravel
          verboseMode: true
          testMode: true
          configPath: "project/pint.json"
          pintVersion: 1.17.2
```

Figura 13 – Workflow para verificar a estilização de código PHP com o uso do Pint.

Fonte: Autoria própria.

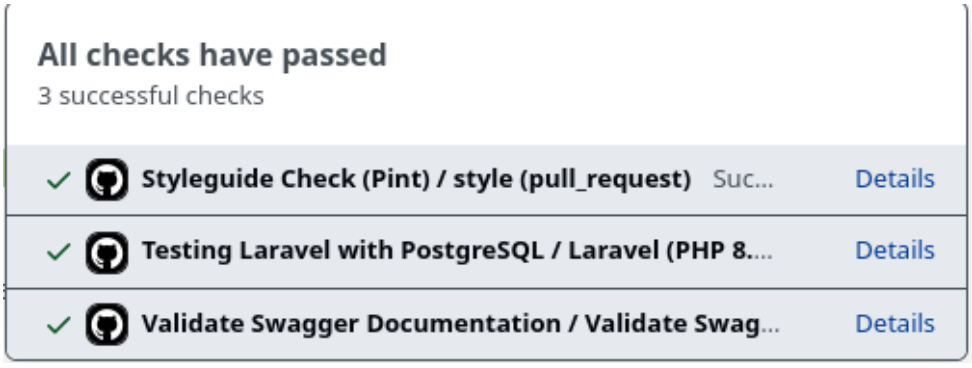


Figura 14 – Exemplo de execução correta de workflows no GitHub.

Fonte: Autoria própria.

```
return (
    <PaperProvider theme={theme}>
      <Stack.Screen
        name="(tabs)"
        />
      </Stack>
    </PaperProvider>
  );
```

Figura 15 – Exemplo de contexto em React, demonstrando a importação do Provider da biblioteca React Native Paper no layout básico da aplicação, de forma que, os componentes dessa biblioteca possam ser chamados em todos os lugares da aplicação.

Fonte: Autoria própria.

```
import axios from 'axios';
import environments from '@/config/environments';
import AsyncStorage from
'@react-native-async-storage/async-storage';

const axiosInstance = () => {
  const instance = axios.create({
    baseURL: environments.apiUrl,
    headers: {
      'Content-Type': 'application/json',
      'Accept': 'application/json',
    },
    withCredentials: true,
    timeout: 5000
  })

  instance.interceptors.request.use(
    async request => {
      const token = await AsyncStorage.getItem('token');

      if (token) {
        request.headers['Authorization'] = `Bearer ${token}`
      }

      return request
    },
    error => {
      return Promise.reject(error)
    },
  )

  return instance;
}

export default axiosInstance;
```

Figura 16 – Instância do Axios com configurações padrão utilizadas no sistema.

Fonte: Autoria própria.

7 RESULTADOS

Neste capítulo, será apresentado o aplicativo desenvolvido, destacando suas telas e funcionalidades como resultado do projeto (detalhes do processo de desenvolvimento e da modelagem do banco de dados do sistema estão disponíveis no Apêndice A, e detalhes da modelagem do banco de dados estão disponíveis no Apêndice B). As telas serão organizadas em duas seções, correspondentes às duas abas disponíveis no aplicativo. Além disso, será demonstrada a modelagem do banco de dados final do sistema, ilustrando a estrutura e organização das tabelas que sustentam as funcionalidades do aplicativo.

7.1 Telas da Aba Área de Usuário

Telas de Autenticação

As primeiras telas apresentadas serão as de autenticação, ou seja, a tela de login e a de cadastro de novo usuário. Na aba *Área do Usuário*, ao acessar o aplicativo pela primeira vez sem um usuário autenticado, o formulário de login será exibido. Nele, o usuário utilizará seu e-mail e senha. Após um login bem-sucedido, uma mensagem de confirmação será exibida na tela.

Para acessar a tela de cadastro de usuário, basta clicar no texto abaixo do formulário de login, que redirecionará para uma tela onde é possível inserir os seguintes dados: nome, e-mail, senha e confirmação de senha. Caso ocorra algum erro de preenchimento, seja no aplicativo (*frontend*) ou no retorno da API, a mensagem de erro correspondente será exibida abaixo do campo afetado. Estas telas descritas estão representadas na Figura 17.

Tela de Gestão de Solicitações

Após a autenticação, o usuário pode acessar a tela de gestão de solicitações, onde estão concentradas a maioria das funcionalidades: edição e exclusão de solicitações, atualização de situação (fechamento), exibição paginada conforme a situação atual e visualização detalhada de cada solicitação.

A listagem de solicitações conta com paginação, e cada item exibido possui um botão que, ao ser clicado, revela as opções disponíveis. A interface dessa tela pode ser vista na Figura 18. Além disso, a tela de detalhes de uma solicitação também pode ser acessada diretamente pelo mapa, ao clicar em um marcador correspondente a uma solicitação.

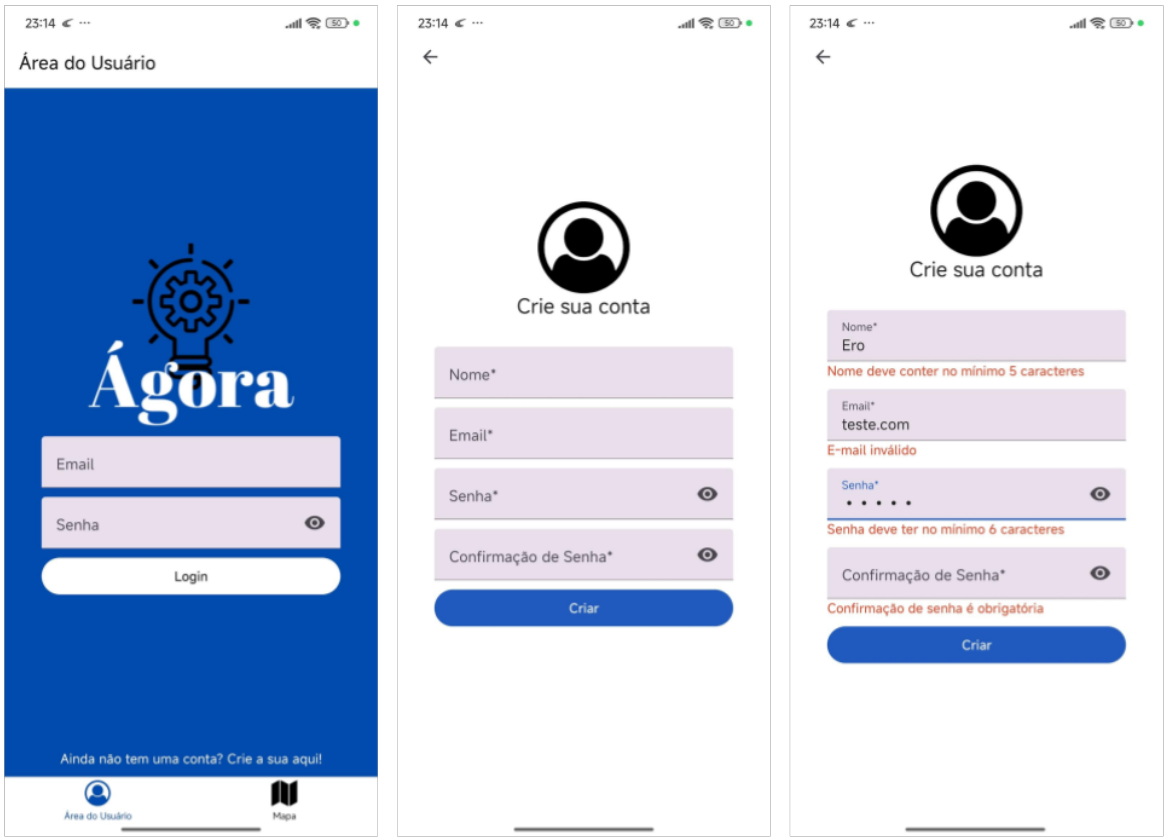


Figura 17 – Capturas de telas que compõem a autenticação do sistema: Tela de login e cadastro de usuário.

Fonte: Autoria própria.

Telas de Atualização de Situação e de Edição de Solicitações

As principais telas desta aba são a tela de edição de solicitação e a tela de atualização de situação.

Na edição de uma solicitação, além de atualizar os dados, o usuário pode adicionar novas imagens e remover as existentes. Caso queira excluir todas as imagens antigas, há um botão específico para isto, porém, é obrigatório o envio de pelo menos uma nova imagem, que servirá como capa da solicitação.

Para atualizar a situação, o usuário deve selecionar a nova etapa por meio de um select e adicionar uma imagem como evidência. Essa imagem será exibida na tela de detalhes da solicitação e registrada em seu histórico. O formulário de atualização de situação, juntamente com o registro da atualização, está ilustrado na Figura 19.

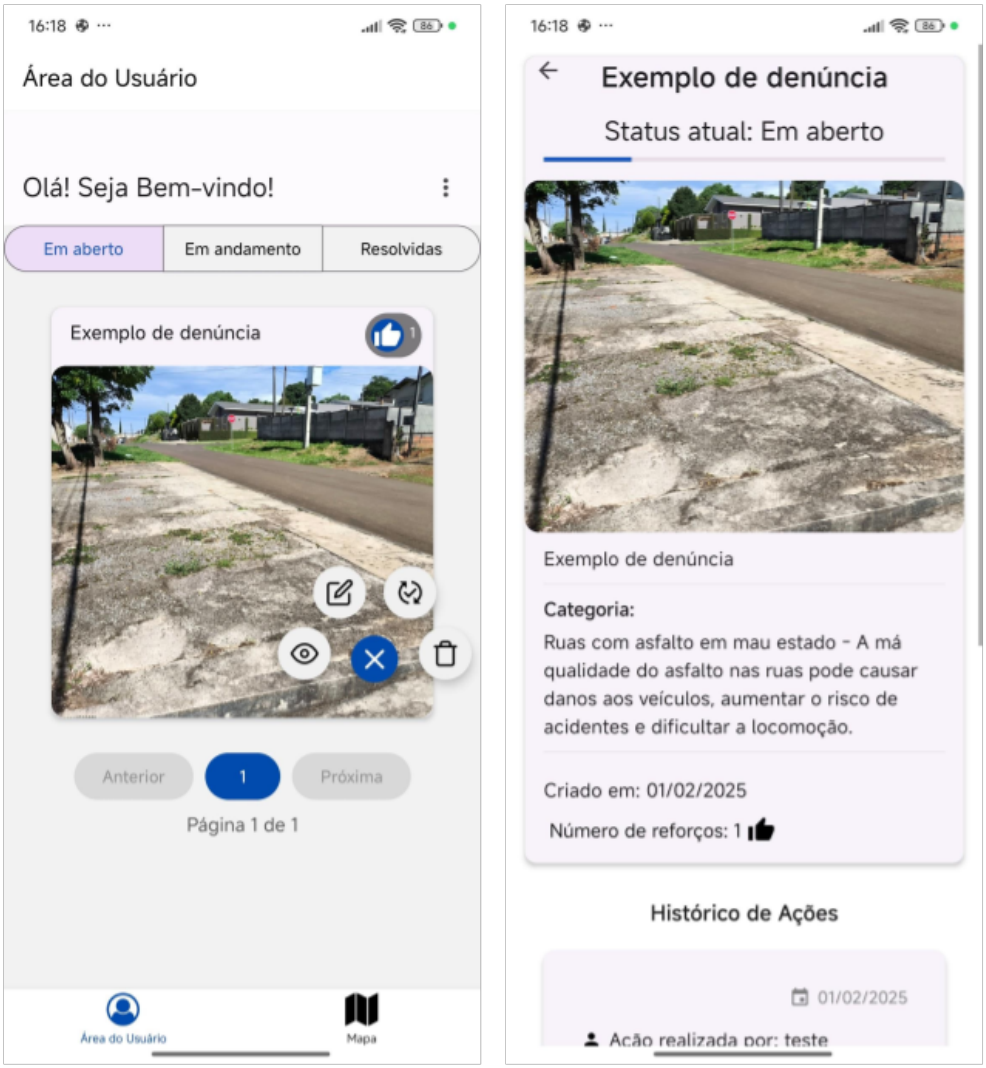


Figura 18 – Captura de tela de aplicativo da tela de listagem de solicitações do usuário atual e tela de exibição de dados de solicitação de forma individual.
Fonte: Autoria própria.

Telas de Gerenciamento de Dados Pessoais e Endereço, com Opção de Logout

Por fim, ainda na aba "Área do Usuário", um usuário autenticado tem acesso às seguintes funcionalidades: adição e atualização de endereço, atualização de dados pessoais e de acesso, e logout. Todas essas opções estão acessíveis por meio de um botão no canto superior direito. O logout apenas encerra a sessão no dispositivo, sem redirecionar para uma tela específica. Já as demais funcionalidades levam o usuário a formulários distintos.

A adição de um endereço à conta não é obrigatória no aplicativo, mas é recomendada para futuras melhorias e funcionalidades. Para cadastrá-lo, o usuário pode pesquisar seu CEP; se tudo ocorrer como esperado, os demais campos serão preenchidos automaticamente. Caso haja algum erro ao consultar as APIs externas, o usuário deverá preencher os campos manualmente.

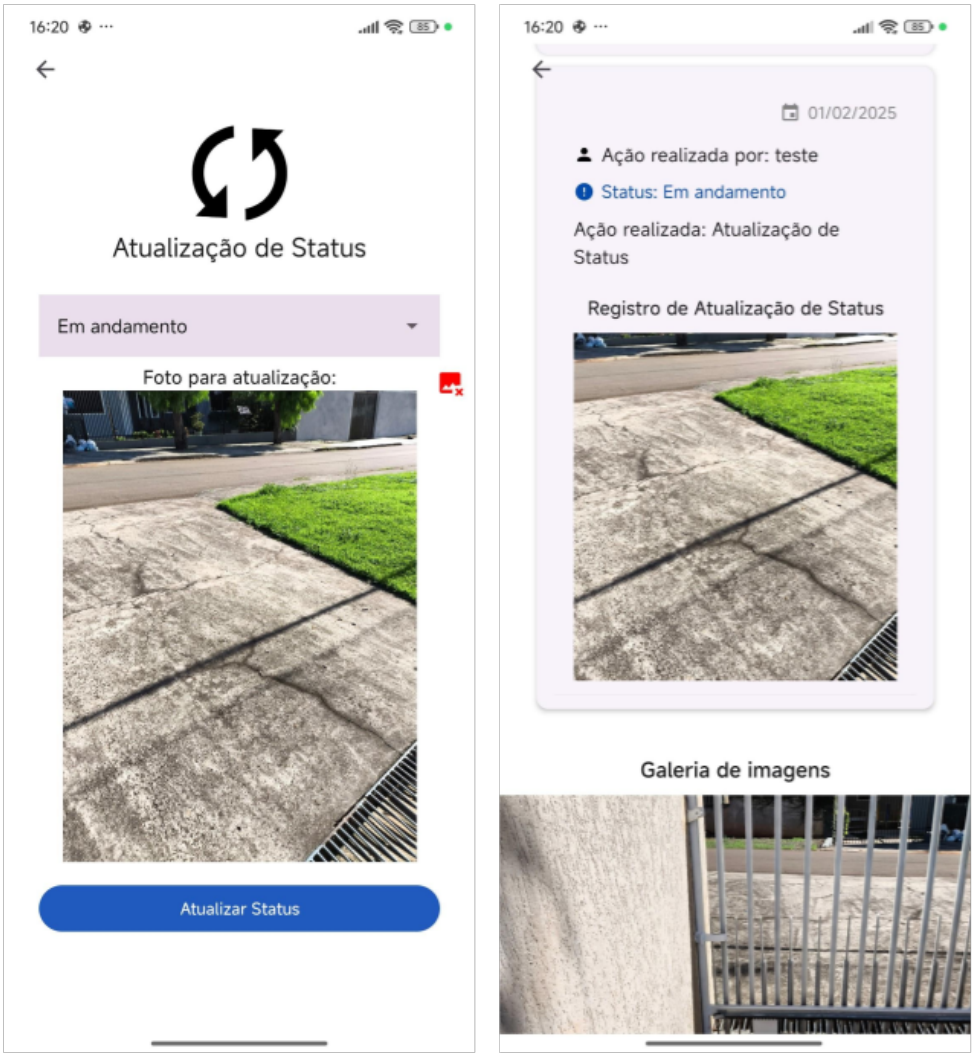


Figura 19 – Captura de tela de aplicativo efetuando a atualização de situação e captura de tela exibindo detalhes de uma solicitação.
Fonte: Autoria própria.

Na tela de atualização de dados pessoais e de acesso, o usuário pode alterar seu nome cadastrado, e-mail e senha. Para modificar a senha, é necessário ativar essa opção e inserir a senha atual, garantindo que a solicitação foi feita pelo próprio usuário e não por um terceiro com acesso ao dispositivo.

As telas de atualização de endereço e dados pessoais estão ilustradas na Figura 20.

7.2 Telas da Aba Mapa

A tela principal do sistema é a aba *Mapa*. Nessa tela, mesmo sem autenticação, o usuário pode visualizar registros de solicitações dentro de um raio de 300 metros de sua localização atual, desde que conceda permissão para o acesso à localização em tempo real. As solicitações exibidas no mapa possuem ícones correspondentes à sua situação atual, facilitando a

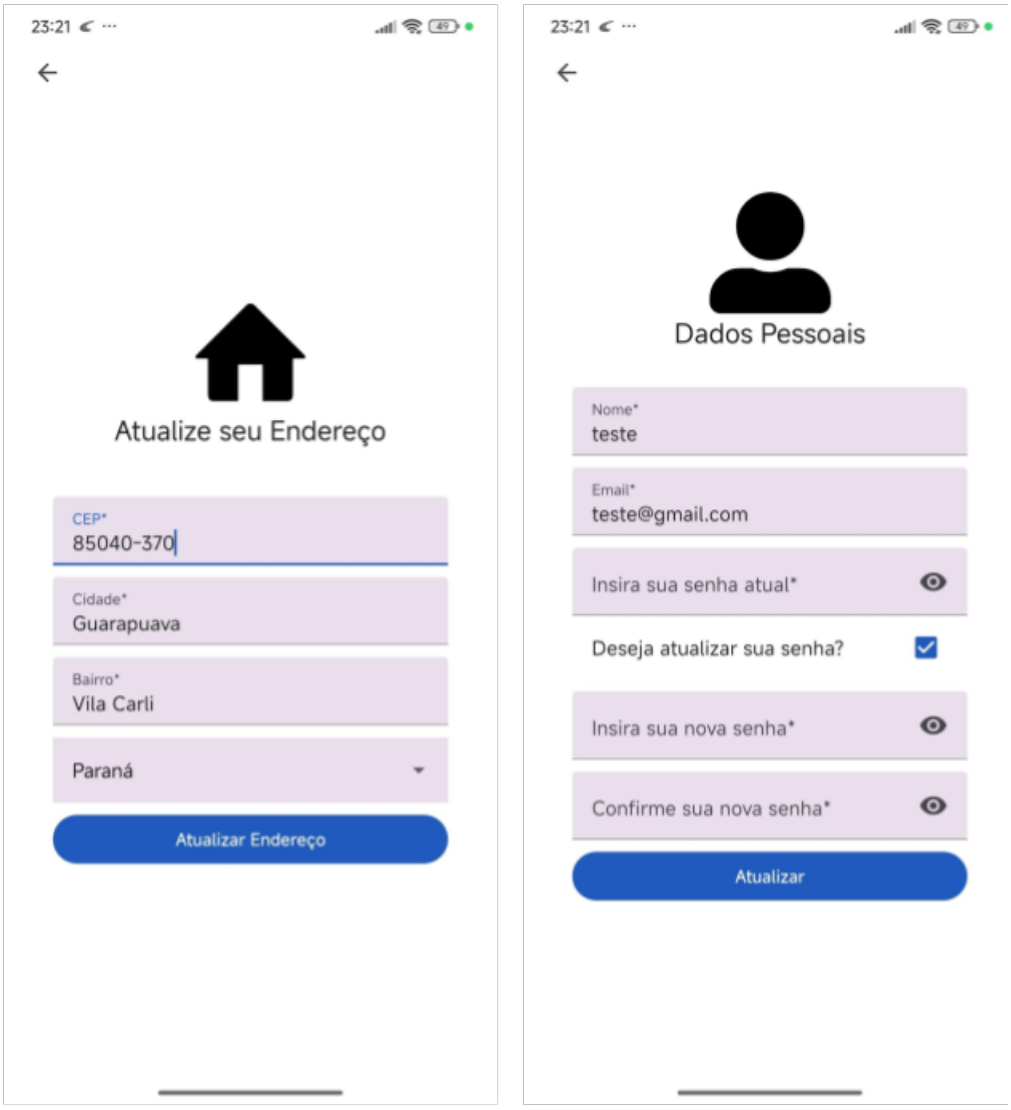


Figura 20 – Captura de tela de aplicativo da tela endereço e atualização de dados pessoais. Fonte: Autoria própria.

identificação da etapa de cada item pelo usuário, mesmo antes de clicar sobre ele. Além disso, a localização atual do usuário é representada por um ícone específico. A Figura 21 demonstra o momento de primeiro acesso, no qual o usuário deve permitir o acesso à localização do seu dispositivo e como o mapa é exibido.

Um botão para cadastro de solicitações está disponível nessa tela, porém, para utilizá-lo, o usuário precisa estar autenticado. Caso contrário, uma mensagem de erro será exibida.

Tela de Cadastro de Solicitações

A partir da tela inicial do mapa, é possível acessar a tela de cadastro de solicitações de manutenção urbana, desde que o usuário esteja autenticado, conforme mencionado na 7.2.

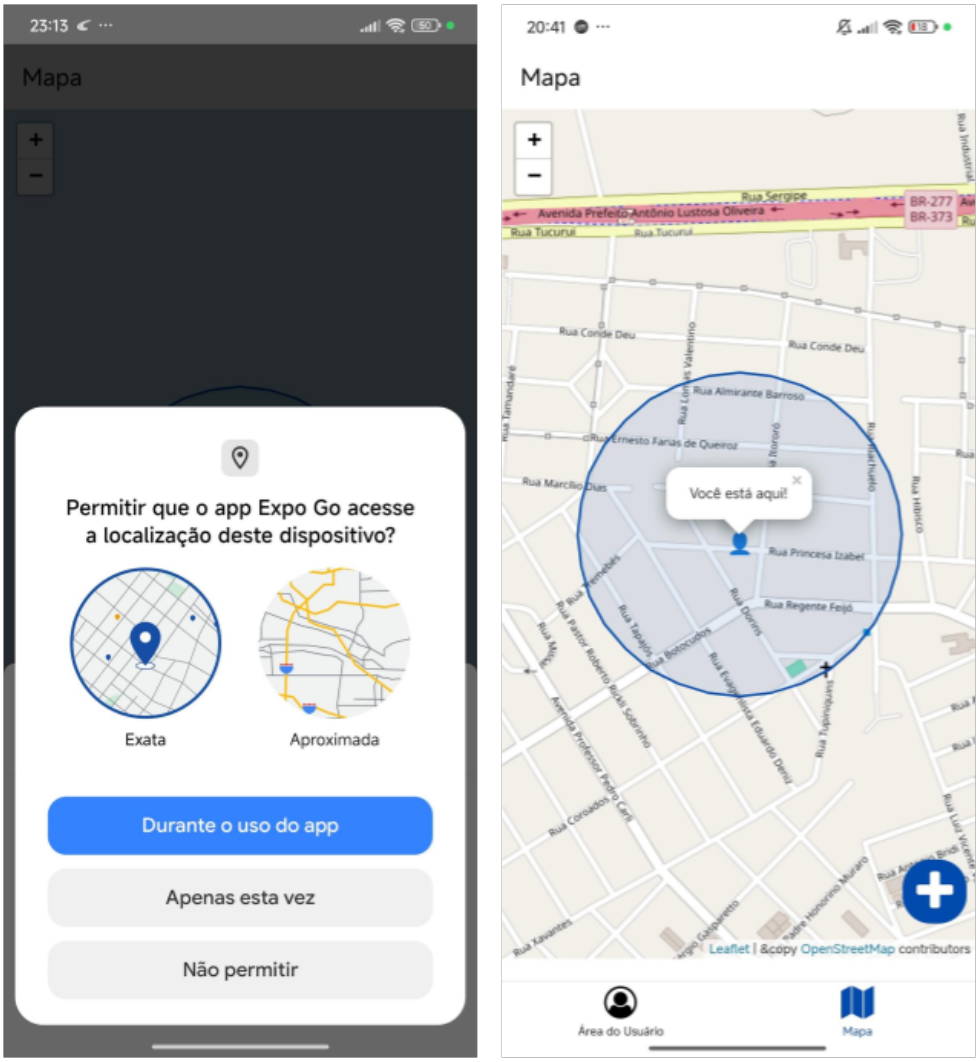


Figura 21 – Captura de tela de aplicativo pedindo permissão para acesso de localização em tempo real e tela de mapa após a permissão concedida.
Fonte: Autoria própria.

Ao acessar essa página, um formulário será exibido para que o usuário insira os dados da denúncia, incluindo título, descrição e a categoria mais adequada. Como o formulário é dividido em etapas, o usuário só poderá avançar para a próxima fase após preencher corretamente todos os campos obrigatórios.

Na segunda etapa, ocorre a captura das imagens. Caso o usuário conceda permissão para acesso à câmera, será possível tirar fotos diretamente pelo dispositivo. Após a captura, ele pode avaliar as imagens e, se considerar que representam bem o problema, prosseguir com a criação da solicitação. Caso contrário, há a opção de excluir as fotos individualmente por meio de um botão ao lado de cada imagem.

O processo de cadastro de uma nova solicitação está ilustrado na Figura 22.

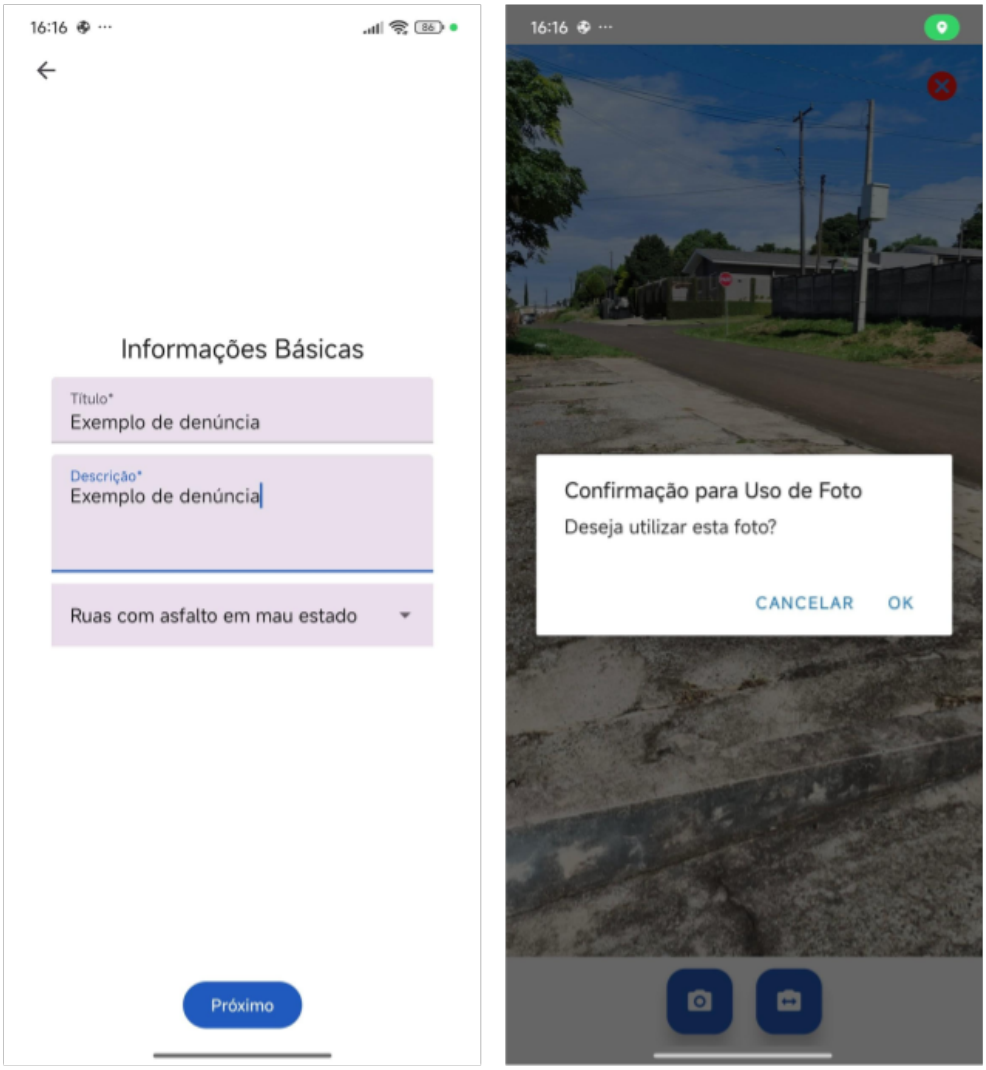


Figura 22 – Captura de tela de aplicativo, exibindo formulário de cadastro de nova solicitação de manutenção urbana e processo de captura de foto via câmera.
Fonte: Autoria própria.

Reforço de Solicitações

Assim como mencionado na subseção 7.1, a tela de detalhes de uma solicitação pode ser acessada a partir do mapa, ao clicar em um marcador correspondente. Esse mesmo fluxo também permite que usuários autenticados realizem o reforço de uma solicitação.

Após clicar em um marcador, um pequeno modal será exibido com a imagem de capa da solicitação, oferecendo três opções: visualizar detalhes, reforçar a solicitação ou simplesmente fechar o modal. O número total de reforços recebidos por uma solicitação é exibido diretamente nesse modal, e o ícone de reforço muda de estado quando a ação é realizada pelo usuário, tornando fácil identificar visualmente quais solicitações ele já reforçou.

Caso o reforço seja realizado com sucesso, uma notificação de confirmação é exibida e o modal é fechado automaticamente. A Figura 23 ilustra esse processo, demonstrando uma solicitação sendo reforçada com sucesso.

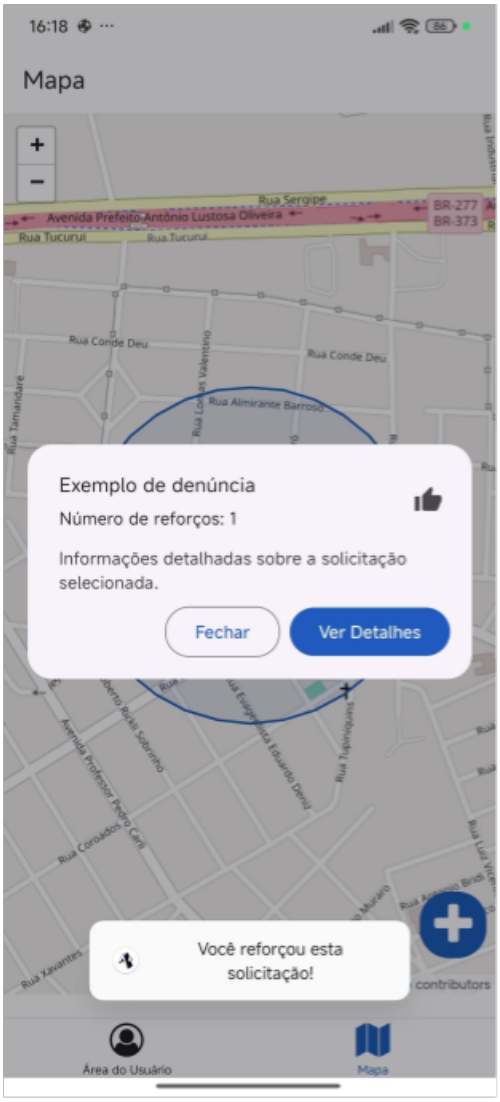


Figura 23 – Captura de tela do aplicativo exibindo uma solicitação reforçada com sucesso. O ícone de reforço está destacado, indicando visualmente ao usuário que a solicitação já recebeu seu apoio.

Fonte: Autoria própria.

8 CONCLUSÕES

Neste capítulo, apresentam-se as conclusões finais do trabalho, bem como suas perspectivas futuras. Apresentando os principais resultados alcançados e a contribuição do sistema desenvolvido. Já na seção 8.1, são sugeridas melhorias e novas funcionalidades que poderão ser implementadas para aprimorar e expandir o projeto.

Este trabalho propôs e descreveu o desenvolvimento de um sistema para consulta e registro de solicitações de manutenção de problemas urbanos, por meio do aplicativo Ágora. O sistema foi desenvolvido utilizando a linguagem de programação PHP com o framework *Laravel* para a API e React Native para o aplicativo móvel. A escolha dessas tecnologias foi baseada nas vantagens que oferecem para o desenvolvimento.

Além disso, foram abordadas as metodologias utilizadas no desenvolvimento do sistema, desde a fase inicial de análise e levantamento de requisitos, passando pela prototipação, até a implementação. Um dos principais objetivos do projeto foi garantir que, ao final do desenvolvimento, o software atenda a todos os seus requisitos funcionais, além de ser robusto, com código limpo e organizado, facilitando futuras atualizações e manutenções.

Com base nos objetivos estabelecidos no início do projeto, conclui-se que o sistema alcançou suas metas principais. Sua implementação pode proporcionar benefícios como maior transparência para o cidadão, permitindo acesso a informações sobre problemas urbanos e as ações tomadas para solucioná-los. Além disso, a solução busca incentivar a participação ativa da população no papel de agente fiscalizador. Em uma sociedade, com cada cidadão fazendo a sua parte, grandes melhorias podem ser alcançadas, como demonstrado de forma lúdica na Figura 24.

8.1 Trabalhos Futuros

Mesmo com a conclusão do trabalho e o cumprimento dos principais objetivos, é importante reconhecer que existem diversas possibilidades de melhorias e novas funcionalidades que podem agregar ao sistema. Entre elas, destaca-se a adição de um sistema de enquetes, permitindo que os usuários participem de votações sobre temas diversos relacionados às suas localizações, como a priorização de problemas urbanos, pesquisas de satisfação, entre outros. Essa funcionalidade pode ampliar o engajamento dos cidadãos, tornando o sistema ainda mais interativo e colaborativo.

Outra melhoria relevante seria a implementação de um sistema de pontos, no qual usuários frequentes, que seguem as regras e utilizam o sistema ativamente, acumulam pontos. Esses pontos poderiam ser utilizados para desbloquear ações adicionais, como a possibilidade de atualizar a situação de solicitações criadas por outros usuários ou até mesmo para obter privilégios de moderador. Essa abordagem ajudaria a garantir que as solicitações sejam validadas

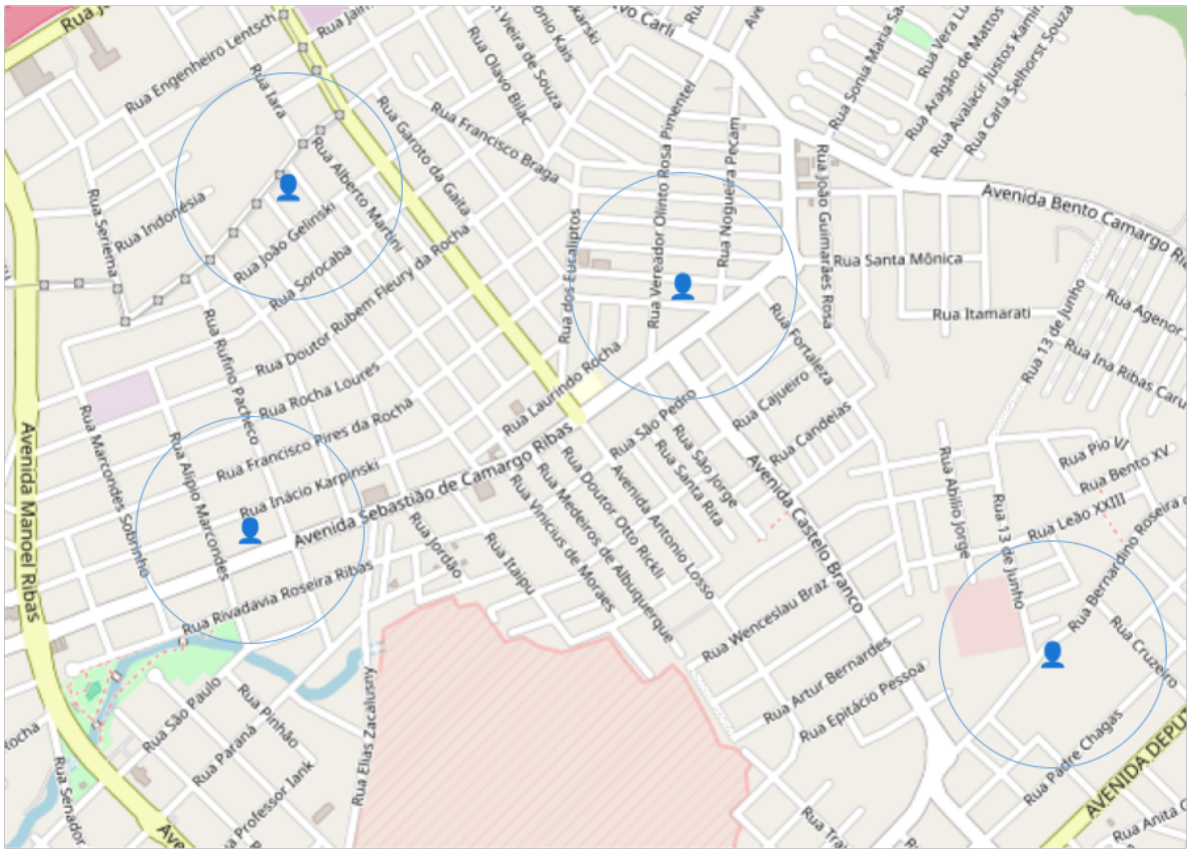


Figura 24 – Imagem lúdica, com captura de tela de mapa do OpenStreetMap, demonstrando a importância da participação dos cidadãos, pois cada um pode ser um agente fiscalizador no local onde vive.

Fonte: Autoria própria.

por usuários comprometidos, assegurando que sejam problemas reais e que o conteúdo esteja adequado.

Além disso, o design do aplicativo poderia ser aprimorado, tendo em vista que as aplicações modernas estão cada vez mais dinâmicas e intuitivas. Melhorias nessa área garantiriam uma experiência de usuário mais fluida e agradável, alinhada com as tendências de usabilidade.

Outro ponto importante a ser considerado no futuro do sistema é a forma de armazenamento de tokens no aplicativo móvel. Embora a abordagem atual, utilizando o AsyncStorage, seja comum, ela apresenta vulnerabilidades. De acordo com a documentação oficial do React Native, ao utilizar o AsyncStorage para armazenar dados, esses são gravados no armazenamento do dispositivo, o que os torna mais suscetíveis a invasões. Assim, a própria documentação recomenda seguir outras abordagens mais seguras, dependendo do sistema operacional em que o aplicativo será utilizado (Android ou IOS) (META, 2025a).

Dessa forma, todas essas possibilidades podem contribuir significativamente para a evolução e aprimoramento do sistema no futuro, proporcionando uma solução mais robusta, segura e interativa, com o potencial de engajar ainda mais os cidadãos e otimizar a gestão de solicitações de manutenção urbana.

REFERÊNCIAS

ACOSTA, J.; GAJDA, K. **Test-driven development**. 2023. Disponível em: https://www.ibm.com/garage/method/practices/code/practice_test_driven_development/.

ATLASSIAN. 2023. Disponível em: https://www.atlassian.com/work-management/knowledge-sharing/documentation/importance-of-documentation.

ATLASSIAN. 2023. Disponível em: https://www.atlassian.com/br/git/tutorials/comparing-workflows/gitflow-workflow.

BEMESTAR, S. 2025. Disponível em: https://www.saudebemestar.pt/pt/blog-saude/qualidade-de-vida/.

CAMP, J. The athenian agora: Excavations in the heart of classical athens. **Introduction**, Thames Hudson, p. 14–19, 1986. Tradução de Gisele A. D. F. Arrunátegui; revisão de Labeca.

CARTA Brasileira: Cidades Inteligentes. 2020. Disponível em: https://www.gov.br/mdr/pt-br/assuntos/desenvolvimento-urbano/carta-brasileira-para-cidades-inteligentes/CartaBrasileiraparaCidadesInteligentes2.pd.

CHAVES, M. 2023. Disponível em: https://dev.to/matheushchaves/como-e-por-que-usar-o-axios-44gh.

DOCKERINC. 2023. Disponível em: https://www.docker.com/why-docker/.

DOCKERINC. 2023. Disponível em: https://docs.docker.com/compose/.

DOCKERINC. 2024. Disponível em: https://docs.docker.com/storage/volumes/.

DOCKERINC. 2024. Disponível em: https://docs.docker.com/guides/docker-concepts/the-basics/what-is-an-image/.

DUARTE, P. 2018. Disponível em: https://paulohincdig.medium.com/o-que-%C3%A9-ddd-c2e5fd7f1687.

EUROPE, W. H. O. R. O. for. Ottawa charter for health promotion, 1986. World Health Organization. Regional Office for Europe, p. 4 p., 1986.

EXPO. 2025. Disponível em: https://docs.expo.dev/versions/latest/sdk/camera/.

EXPO. 2025. Disponível em: https://docs.expo.dev/versions/latest/sdk/filesystem/.

FORMIK. 2025. Disponível em: https://formik.org/docs/overview.

GITHUB. 2023. Disponível em: https://docs.github.com/en/issues/planning-and-tracking-with-projects/learning-about-projects/about-projects.

GITHUB. 2025. Disponível em: https://docs.github.com/pt/actions/writing-workflows/about-workflows.

JACOBI, P. A percepção dos problemas ambientais urbanos em são paulo. **Lua Nova: Revista de Cultura e Política**, 2013.

JUHL, J.; SøRENSEN, A. S. **Prototyping: Getting the design right**. 2023. Disponível em: https://www.ibm.com/blogs/nordic-msp/prototyping/.

LARAVEL. 2023. Disponível em: https://laravel.com/docs/10.x#why-laravel.

LARAVEL. 2023. Disponível em: https://laravel.com/docs/10.x/testing#introduction.

LARAVEL. 2023. Disponível em: https://laravel.com/docs/10.x/pint.

LARAVEL. 2024. Disponível em: https://laravel.com/docs/11.x/installation#creating-a-laravel-project.

LARAVEL. 2024. Disponível em: https://laravel.com/docs/11.x/seeding.

LARAVEL. 2024. Disponível em: https://laravel.com/docs/11.x/structure#the-root-app-directory.

LARAVEL. 2025. Disponível em: https://laravel.com/docs/11.x/sanctum.

LARAVEL. 2025. Disponível em: https://laravel.com/docs/11.x/validation.

LARAVEL. 2025. Disponível em: https://laravel.com/docs/11.x/artisan#introduction.

LONGEN, A. S. **O Que é GitHub, Para Que Serve e Como Usar**. 2023. Disponível em: https://www.hostinger.com.br/tutoriais/o-que-github.

MARQUES, V. 2025. Disponível em: https://www.todamateria.com.br/urbanizacao/.

META. 2023. Disponível em: https://reactnative.dev/.

META. 2025. Disponível em: https://reactnative.dev/docs/security.

META. 2025. Disponível em: https://reactnative.dev/docs/asyncstorage.

MIRO. 2023. Disponível em: https://miro.com/pt/diagrama/diagrama-entidade-relacionamento/.

NPM. 2025. Disponível em: https://www.npmjs.com/package/burnt/v/0.1.0.

POSTGRESQL. 2023. Disponível em: https://www.postgresql.org/about/.

SANTANA, B. **Eleve Seus Designs: O Que é Figma e Como Usá-lo Corretamente!** 2023. Disponível em: https://www.hostinger.com.br/tutoriais/figma-o-que-e#:~:text=Aprendemos%20que%20o%20Figma%20%C3%A9,aplicativos%20e%20sites%2C%20por%20exemplo.

SOMMERVILLE, I. **Engenharia de Software**. [*S.l.*]: Pearson Education do Brasil Ltda., 2011. 57 p.

SOMMERVILLE, I. **Engenharia de Software**. [*S.l.*]: Pearson Education do Brasil Ltda., 2011. 59 p.

SPATIE. 2025. Disponível em: https://spatie.be/docs/laravel-medialibrary/v11/introduction.

STEINFELD, G. 2024. Disponível em: https://developer.ibm.com/articles/5-steps-of-test-driven-development/.

Tribunal Superior Eleitoral. **Cidadão tem papel fundamental na fiscalização de seus representantes**. 2017. Disponível em: https://www.tse.jus.br/comunicacao/noticias/2017/Fevereiro/cidadao-tem-papel-fundamental-na-fiscalizacao-de-seus-representantes.

VIEIRA, L. F. 2023. Disponível em: https://aprovatotal.com.br/principais-problemas-urbanos/.

YUP. 2025. Disponível em: https://github.com/jquense/yup?tab=readme-ov-file.

ZUSTAND. 2025. Disponível em: https://zustand.docs.pmnd.rs/getting-started/introduction.

APÊNDICE A – Desenvolvimento do Sistema Ágora

Neste apêndice, serão descritos todos os ciclos de desenvolvimento, desde o início do sistema até sua conclusão, incluindo as mudanças ocorridas no planejamento. Todo o desenvolvimento seguiu a metodologia descrita na seção 4.2 e ocorreu no período de 17/01/2024 a 07/01/2025.

Durante o processo, alguns atrasos inesperados impactaram o cronograma, como problemas de compatibilidade entre a biblioteca utilizada e funcionalidades de mapa no aplicativo móvel. Para solucionar essa questão, foi necessário substituir o gerenciador de pacotes. Essa atualização está detalhadamente descrita na subseção A.3.2.

A.1 Ciclo 1

O desenvolvimento do sistema teve início pela implementação das funcionalidades de autenticação, fundamentais para o seu funcionamento. As primeiras telas criadas foram as de login, cadastro de usuários e demais interfaces relacionadas à autenticação. Para isso, foi necessário o uso de bibliotecas externas, que serão detalhadas nesta seção.

A.1.1 Autenticação de Usuários

Além das modificações estruturais do projeto elaboradas previamente, foram implementados os primeiros endpoints e funcionalidades do sistema. Na API, foram desenvolvidas as funções de login, logout e o cadastro de novos usuários. Para gerenciar o sistema de autenticação, utilizou-se uma biblioteca que, nas versões mais recentes do framework, foi integrada de forma padrão: o Sanctum[1]. Esta biblioteca consta na documentação e fornece um sistema de autenticação leve, permitindo que cada usuário gere vários tokens, podendo ser utilizada em Single Page Applications (SPA)'s, API's e aplicativos mobile (LARAVEL, 2025a).

Então, no aplicativo, foram desenvolvidas as primeiras telas: formulário de login, cadastro de usuários e a tela do usuário logado. Com a autenticação já implementada na API, foi possível avançar e implementar as mesmas funcionalidades no aplicativo. Para facilitar o desenvolvimento da autenticação e o gerenciamento de tokens, foram utilizadas duas bibliotecas em conjunto:

1. Zustand[2], conforme descrita em sua documentação, é um gerenciador de estados utilizado para armazenar tokens durante a execução da aplicação (ZUSTAND, 2025);

2. AsyncStorage[3], segundo a documentação oficial do React Native, exerce uma função similar ao Local Storage [4] presente nos navegadores. Embora tenha sido descontinu-

[1] https://laravel.com/docs/11.x/sanctum
[2] https://zustand-demo.pmnd.rs/
[3] https://docs.expo.dev/versions/latest/sdk/async-storage/
[4] https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage

ado, a comunidade desenvolveu uma versão própria, atualmente recomendada pela documentação oficial do React Native (META, 2025b).

A integração dessas duas bibliotecas permitiu implementar o gerenciamento de tokens com armazenamento local no dispositivo, garantindo que a sessão do usuário fosse mantida mesmo após o fechamento da aplicação. A função que realiza o login e armazena o token no armazenamento local do dispositivo está demonstrada na Figura 25.

```
export const
 useAuthStore = create<AuthState>((set) => ({
  token: null,
  setToken: (token) => {
    set({ token: token });
  },
  login: async (credentials) => {
    try {
      const response = await
 axiosInstance().post<LoginResponse>(apiRoutes.
auth.login, {
        email: credentials.email,
        password: credentials.password,
      });
      if (response.data.token) {
        set({ token: response.data.token });
        await AsyncStorage.setItem('token'
, response.data.token);
        successToast({ title:
'Sessão iniciada! Bem-vindo!' });
      } else {
        throw new Error('Token is not valid');
      }
    } catch (error: any) {
      errorToast({ title:
'Falha ao iniciar sessão. Tente novamente.' });
    }
  },
```

Figura 25 – Store criado utilizando o Zustand juntamente com o AsyncStorage.
Fonte: Autoria própria.

As primeiras telas desenvolvidas foram as de login e cadastro de novos usuários, nas quais os formulários iniciais foram implementados. Esses formulários foram desenvolvidos com o suporte das bibliotecas Formik e Yup, estabelecendo um padrão reutilizável adotado nos demais formulários simples do sistema. Para isso, foi criado um arquivo específico para centralizar as validações, que era importado diretamente no componente de cada formulário, promovendo uma organização mais eficiente e facilitando a manutenção do código.

Na Figura 26 está exibido o arquivo de validação utilizado no formulário de login. Nesse arquivo, os campos de entrada são definidos com suas respectivas regras de validação e mensagens de erro, garantindo um controle centralizado e consistente sobre os dados enviados.

```
import * as Yup from 'yup';

const LoginValidation = Yup.object().shape({
  email: Yup.string()
    .email('E-mail inválido')
    .required('Campo e-mail é obrigatório')
    .trim(),

  password: Yup.string()
    .required('Campo senha é obrigatório')
    .min(6, 'A senha deve ter pelo menos 6 caracteres')
    .max(20, 'A senha deve ter no máximo 20 caracteres')
    .trim(),
});

export default LoginValidation;
```

Figura 26 – Classe que implementa as validações Yup para validar dados de entrada para o login.
Fonte: Autoria própria.

Assim, foi possível desenvolver o formulário de login, a utilização do Formik juntamente com o Yup é exibida na Figura 27, demonstrando a forma própria da biblioteca em trabalhar com formulários e exibição de erros no lado do cliente. Além dessas adições, neste ciclo, optou-se pela biblioteca padrão de notificações, Burnt (https://github.com/nandorojo/burnt), mencionada no site do Node Package Manager (NPM) como uma solução de notificações multiplataforma que utiliza elementos nativos (NPM, 2025). Essa escolha proporcionou uma integração mais eficiente e consistente das notificações na aplicação.

```
<Formik
        enableReinitialize
        initialValues={{
          email: '',
          password: '',
          keepSession: false,
        }}
        validationSchema={LoginValidation}
        onSubmit={(values) => login(values)}
      >
        {({ handleChange, handleSubmit, values, errors, touched, isValid, setFieldValue }) => (
          <View style={styles.form}>
            <TextInput
              style={styles.space}
              label="Email"
              placeholder="Seu email"
              value={values.email}
              onChangeText={handleChange('email')}
            />
            {(errors.email && touched.email) && <FormError errorMessage={errors.email} />}

            <PasswordInput
              label="Senha"
              value={values.password}
              placeholder="Sua senha"
              onChangeText={handleChange('password')}
            />
            {(errors.password && touched.password) && <FormError errorMessage={errors.password} />}

            <Button
              style={styles.button}
              labelStyle={styles.buttonText}
              onPress={(e: any) => handleSubmit(e)}
              mode="contained"
              disabled={!isValid}
            >
              Login
            </Button>
          </View>
        )}
      </Formik>
```

Figura 27 – Formulário de login utilizando as bibliotecas Formik e Yup.
Fonte: Autoria própria.

A.2 Ciclo 2

Nesta seção, serão abordadas as funcionalidades relacionadas ao cadastro de endereços dos usuários, bem como as melhorias realizadas na estruturação do banco de dados durante esta fase do desenvolvimento.

A.2.1 Cadastro de Endereço de Usuários

Com a implementação da autenticação de usuários, foi possível avançar para a criação dos primeiros *endpoints* autenticados da API. Para o cadastro de um novo usuário, são necessárias apenas suas credenciais, que serão utilizadas para acessar a aplicação. No entanto, para obter informações mais detalhadas sobre os usuários que utilizarão o sistema, foram desenvolvidas funcionalidades para exibição e cadastro/edição de endereço. Dessa forma, cada usuário

pode inserir seu bairro e cidade de residência, permitindo um melhor entendimento do perfil dos usuários e possibilitando melhorias futuras, como a personalização de serviços conforme a localização.

A.2.2 Evolução da Estrutura do Banco de Dados

Para aprimorar a eficiência das operações de busca, consulta e cadastro na aplicação, foi realizada uma reestruturação no banco de dados. Entre as alterações implementadas, destaca-se a criação de uma tabela intermediária entre as entidades de usuários e solicitações, com o objetivo de eliminar campos redundantes. Além disso, a relação entre usuários e endereços foi reconfigurada para otimizar o armazenamento e o acesso às informações. As tabelas mencionadas nessa atualização estão demonstradas na Figura 28.

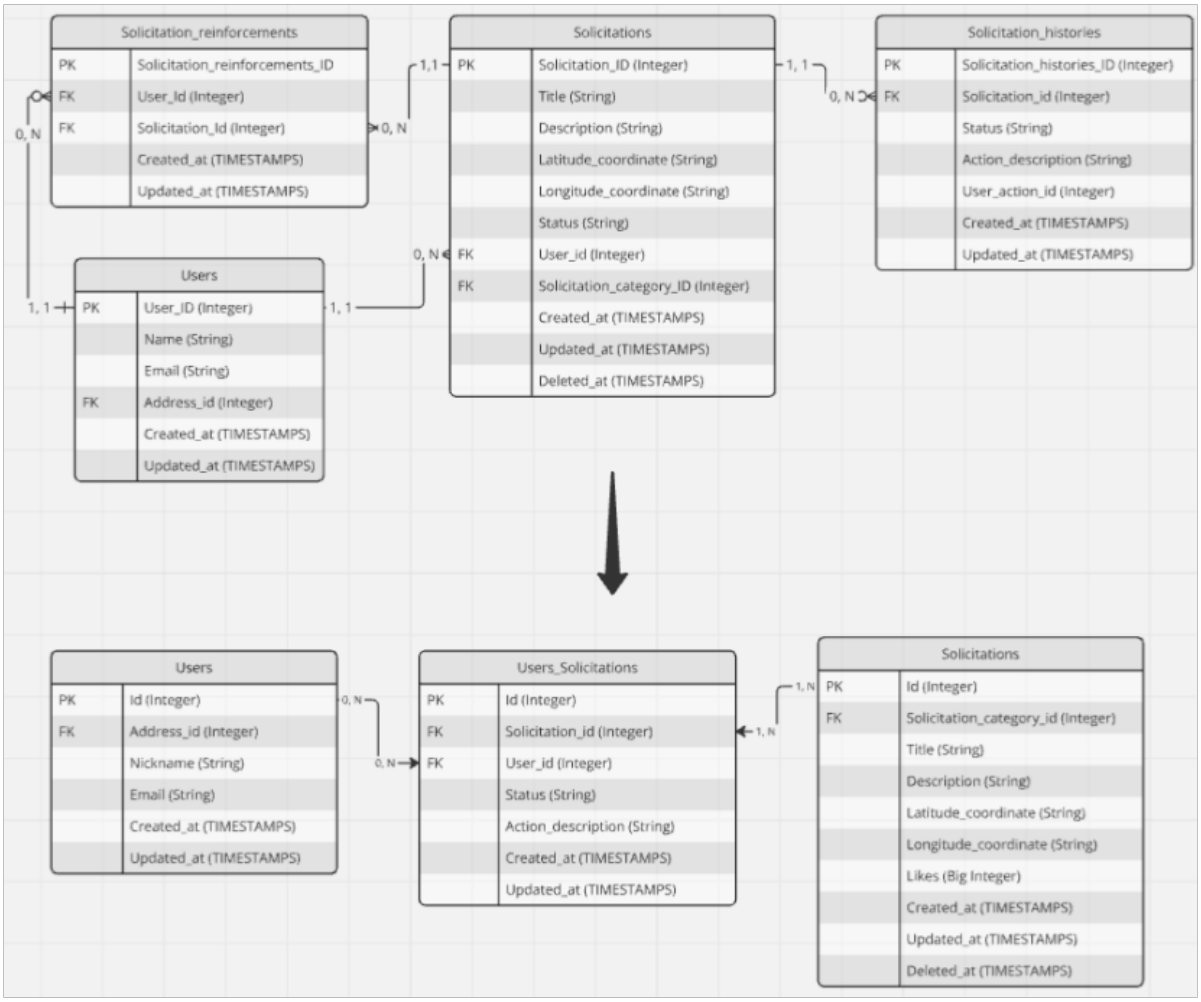


Figura 28 – Diagrama do banco de dados do sistema, exibindo as tabelas em sua primeira versão, de acordo com o MER produzido como resultado na análise inicial e as tabelas abaixo após a atualização.

Fonte: Autoria própria.

A.3 Ciclo 3

Durante este terceiro ciclo de desenvolvimento, o sistema passou por uma mudança significativa que não havia sido inicialmente planejada. No entanto, devido a atualizações e ao versionamento das bibliotecas utilizadas, essa adaptação tornou-se necessária para garantir a compatibilidade e o bom funcionamento do aplicativo. Além dessas atualizações, esta seção também aborda o processo de captura da localização do usuário, que foi implementado por meio do uso do GPS do dispositivo. Esse recurso permite que o sistema obtenha a posição geográfica do usuário em tempo real, tornando a experiência mais dinâmica.

A.3.1 Integração com o *GPS* para Captura de Localização Atual

Uma parte essencial do aplicativo é o acesso à localização do usuário em tempo real via GPS. Para implementar essa funcionalidade, o Expo oferece uma biblioteca simples e fácil de usar chamada Location [5]. Conforme a documentação, para obter a localização atual do dispositivo móvel via GPS, é necessário solicitar autorização do usuário. Após a permissão ser concedida, é possível acessar em tempo real a latitude e longitude do aparelho. Para gerenciar o acesso a esses dados, foi utilizado um Context personalizado, destinado a administrar os dados de localização em tempo real do usuário e a controlar as permissões de acesso. A implementação desse Context está ilustrada na Figura 29.

Outra funcionalidade incorporada ao sistema foi a tela de gestão de endereços no aplicativo, permitindo que o usuário visualize o endereço cadastrado e realize sua edição, caso necessário.

A.3.2 Problemas de Compatibilidade

Inicialmente, o gerenciador de pacotes adotado para o desenvolvimento do aplicativo foi o NPM [6], e todo o código até aquele ponto havia sido implementado utilizando essa ferramenta. No entanto, ao tentar instalar uma biblioteca para integração de mapas com o Leaflet, surgiram problemas de compatibilidade. Por ser uma biblioteca mais antiga, ela apresentava conflitos com outras dependências já utilizadas no projeto. Como as opções gratuitas de bibliotecas para mapas são limitadas, várias alternativas foram testadas, mas todas impunham restrições significativas em seu uso de forma gratuita. Após múltiplas tentativas sem sucesso, decidiu-se explorar a possibilidade de alterar o gerenciador de pacotes para tentar viabilizar o uso da biblioteca originalmente planejada [7].

[5] https://docs.expo.dev/versions/latest/sdk/location/
[6] https://www.npmjs.com/
[7] https://github.com/Dean177/expo-leaflet

Como solução, optou-se pela substituição do NPM pelo Yarn [8]. Essa mudança permitiu a instalação bem-sucedida da biblioteca Expo-leaflet, viabilizando a integração de um mapa funcional no aplicativo. Apesar de impactar uma área crítica do sistema, a transição foi concluída sem gerar conflitos no projeto.

A.4 Ciclo 4

Neste ciclo, iniciou-se o processo de gerenciamento das solicitações de manutenção urbana no sistema Ágora. Para assegurar que os dados essenciais estivessem disponíveis desde o início, foram criadas Seeders para o cadastro das categorias utilizadas na criação das solicitações. Além disso, foi implementada a funcionalidade de cadastro de novas solicitações, permitindo que os usuários preenchessem um formulário com as informações necessárias, incluindo a captura de fotos diretamente pela câmera do dispositivo. Também foi desenvolvida a funcionalidade de reforço, permitindo que os usuários expressassem apoio a solicitações já cadastradas, incentivando a colaboração e o engajamento com o sistema.

A.4.1 Registro de Categorias através de Seeders

Previamente à implementação das funcionalidades de gestão das solicitações de manutenção urbana, foi necessário definir e popular as possíveis categorias que essas solicitações poderiam adotar. Considerando que o sistema, em sua versão atual, não dispõe de um painel administrativo, a criação dessas categorias, que servem como opções no sistema, foi realizada por meio da implementação de Seeders.

Após a inserção das categorias no banco de dados, foi desenvolvido um novo endpoint na API para retornar esses itens registrados no banco de dados. Para que, desta forma, os usuários possam selecioná-los no momento de registrar uma nova solicitação.

A.4.2 Cadastro de Solicitações de Manutenção Urbanas

Com a tabela de categorias populada, foram desenvolvidos os endpoints responsáveis pela criação de solicitações de manutenção urbana, assegurando a validação dos dados de entrada tanto no lado cliente (aplicativo móvel) quanto no nível da API. Devido à complexidade do cadastro de solicitações, que envolve tanto campos de texto quanto o envio de imagens, optou-se por implementar um formulário dividido em etapas. Na primeira etapa, o usuário preenche as informações básicas da solicitação, como título, descrição e categoria. Já na segunda etapa, as imagens capturadas pela câmera do dispositivo são enviadas, permitindo a inclusão visual necessária para completar a solicitação.

[8] https://classic.yarnpkg.com/en/

Para a validação dos dados na API, foram empregadas as Requests do *Laravel*, que, conforme descrito na documentação oficial do framework, disponibilizam uma ampla variedade de regras de validação pré-definidas (LARAVEL, 2025b). A Request implementada para validar os dados das solicitações de manutenção urbana está apresentada na Figura 30.

Após a validação inicial realizada via Request, os dados de entrada foram transferidos para um DTO, como ilustra a Figura 31. Em seguida, para a criação efetiva do registro da solicitação de manutenção urbana, foi desenvolvida uma classe denominada Action, encarregada de executar essa tarefa. Essa classe está detalhada na Figura 32.

Para validar o correto funcionamento da Action, foi desenvolvido um teste unitário independente do banco de dados, com o objetivo de verificar se o comportamento da classe está conforme o esperado. O teste implementado está ilustrado na Figura 33. Da mesma forma, todas as Actions consideradas essenciais para o sistema foram testadas de forma unitária.

No aplicativo, identificou-se a necessidade de organizar os arquivos responsáveis por realizar as requisições para a API. Para atender a essa demanda e promover uma arquitetura mais modular e manutenível, foram criados os chamados Services. Esses Services são classes específicas que encapsulam a lógica de comunicação com a API, centralizando as requisições e simplificando o tratamento das respostas recebidas.

Cada Service foi projetado com o objetivo de realizar requisições à API, processar as respostas e, com base no status HTTP retornado, determinar as ações apropriadas. Caso o status HTTP indique sucesso, o aplicativo retorna uma mensagem positiva ao usuário final, confirmando que a operação foi realizada com êxito. Por outro lado, em caso de erro, o aplicativo informaria que ocorreu uma falha durante a operação.

Esse formato de implementação não apenas organiza o código, mas também garante uma camada de abstração que facilita o reaproveitamento. Um exemplo de uma função implementada dentro de um Service pode ser observado na Figura 34, onde é demonstrado o processo de criação de uma solicitação de manutenção urbana, ilustrando como a lógica foi encapsulada para atender às necessidades do aplicativo.

A.4.3 Captura de Imagens via Aplicativo Mobile e Envio para a API

Quando se trata do envio de imagens em um aplicativo mobile, duas opções são comuns: a captura de imagens utilizando a câmera do dispositivo ou o envio de imagens/arquivos já existentes na galeria ou no armazenamento do aparelho. No sistema Ágora, a estratégia adotada foi a captura de imagens em tempo real. Essa abordagem visa garantir que o usuário registre uma situação real que esteja presenciando, evitando o envio de imagens genéricas obtidas na internet ou de situações das quais o usuário não tenha conhecimento direto, o que poderia comprometer a acurácia das informações.

Para integrar a funcionalidade de captura de imagens e o acesso à câmera do dispositivo, o Expo disponibiliza a biblioteca nativa Expo Camera [9]. Conforme descrito na documentação oficial, além de realizar as etapas iniciais de configuração no projeto Expo, é necessário também solicitar permissão do usuário para utilizar a câmera do dispositivo. Com a permissão concedida, tanto a câmera frontal quanto a traseira podem ser utilizadas para a captura de imagens (EXPO, 2025a). Para exercer essas funções no aplicativo do sistema Ágora, foi criado um componente que permite a captura de imagens utilizando ambas as câmeras do dispositivo. Após a captura, um alerta na tela solicita a confirmação do usuário sobre o uso da imagem obtida. Caso a foto não tenha ficado com uma qualidade satisfatória ou não represente bem a situação desejada, o usuário pode descartá-la e tirar uma nova antes de fechar a câmera. Este processo de remoção de imagens capturadas pode ser realizado da mesma forma através do formulário de cadastro.

Com as imagens capturadas, é necessário enviá-las ao sistema, considerando que os arquivos são armazenados temporariamente no dispositivo como imagens. A abordagem mais adequada para realizar esse envio é utilizando um pacote para manipulação de arquivos chamado FileSystem [10]. Conforme descrito em sua documentação, a instalação dessa biblioteca possibilita a execução de diversas funcionalidades, sendo a mais relevante para o aplicativo a função *asyncUpload*, que permite realizar o upload de arquivos de forma assíncrona (EXPO, 2025b). Com base nessa funcionalidade, foi desenvolvida uma classe responsável por implementar o envio de arquivos. Essa classe foi utilizada em todos os contextos onde há necessidade de realizar uploads. A implementação da classe está ilustrada na Figura 35.

A.4.4 Gerenciamento de Imagens na API

Com as imagens sendo enviadas do aplicativo para a API, foi necessário armazená-las e relacioná-las com os modelos criados. Para isto, optou-se pelo uso de uma biblioteca chamada *laravel-medialibrary*[11], desenvolvida pela empresa Spatie. Conforme descrito na documentação oficial da Spatie, essa biblioteca facilita a adição de imagens e sua associação a modelos, permitindo a criação de coleções de imagens e arquivos, entre outras funcionalidades, criando tabelas no banco de dados para gerenciar as relações com os modelos (SPATIE, 2025).

Para integrar essa biblioteca ao sistema, foi necessário configurar o modelo e criar uma função denominada 'registerMediaCollections'. Nessa função, foram definidas as coleções, especificados os comportamentos desejados (como o suporte a múltiplos arquivos) e configurados os formatos de arquivo aceitos. A implementação dessa funcionalidade segue o exemplo apresentado na Figura 36.

[9] https://docs.expo.dev/versions/latest/sdk/camera/
[10] https://docs.expo.dev/versions/latest/sdk/filesystem/
[11] https://spatie.be/docs/laravel-medialibrary/v11/introduction

Após a configuração, foi criado um endpoint exclusivo para a adição de imagens, responsável por receber os arquivos enviados pelo usuário e o identificador da solicitação à qual essas imagens devem ser vinculadas. Com o arquivo recebido e o modelo correspondente, uma função é utilizada para adicionar a imagem à coleção desejada, conforme exemplificado na Figura 37. Assim, ao carregar a relação 'media' nos modelos, tornou-se possível acessar e incorporar as funcionalidades associadas às coleções de imagens, inclusive nos retornos das consultas realizadas pela API.

Como o processo de cadastro na API é composto de etapas, foi criada uma classe que executa mais de uma Action em sequência, e na nomenclatura adotada foi chamada de Strategy. Então, pode-se criar os primeiros registros válidos com imagens no sistema.

A.4.5 Reforço de Solicitação de Manutenção Urbana

Com o cadastro de novas solicitações funcionando corretamente, foi possível dar sequência à implementação de outra funcionalidade prevista na lista de requisitos: a funcionalidade de reforço. Essa funcionalidade permite que os usuários expressem seu apoio a uma solicitação já criada. O número de reforços de uma solicitação reflete quantos usuários passaram pelo local e identificaram o mesmo problema ocorrido anteriormente.

Para implementar essa funcionalidade, utilizou-se o mapa, que já exibia os itens cadastrados, como uma base para interação. Ao clicar em um desses itens, um modal é exibido com informações básicas sobre a solicitação, mas sem detalhes completos. Caso o usuário queira mais informações, como todas as imagens e o histórico completo da solicitação, ele tem a opção de acessá-las. No entanto, se o objetivo for apenas manifestar apoio, ele pode deixar seu reforço diretamente no modal. As solicitações que já foram previamente reforçadas pelo usuário são destacadas visualmente, permitindo que ele veja facilmente onde já contribuiu e, caso deseje, retire seu reforço a qualquer momento.

Essa implementação contou com a criação de um novo endpoint na API, além de um novo Service no aplicativo, responsável por fazer a requisição, garantindo a comunicação entre o usuário e o sistema de forma eficiente e ágil.

A.5 Ciclo 5

Neste ciclo de desenvolvimento, foram implementadas diversas funcionalidades essenciais para o aprimoramento do sistema. Entre os principais avanços, destacam-se as melhorias na gestão de usuários, incluindo a possibilidade de atualização dos dados pessoais e de acesso, oferecendo mais flexibilidade e controle aos próprios usuários. Além disso, foi integrado um sistema robusto de permissões e perfis na API, permitindo uma gestão mais segura e eficiente dos níveis de acesso e das funcionalidades disponíveis para cada tipo de usuário, com

possibilidade de futuras atualizações. Por fim, foi desenvolvida a funcionalidade de exclusão de solicitações de manutenção urbana, proporcionando maior controle e organização na gestão dessas solicitações por meio do aplicativo.

A.5.1 Atualização de Dados Pessoais e de Acesso

Uma das funcionalidades adicionadas neste ciclo foi a atualização de dados pessoais e de acesso. Com essa melhoria, o usuário pode modificar suas informações, como nome, e também atualizar suas credenciais de login, incluindo e-mail e senha.

Para garantir a segurança desse processo, é necessário que o usuário informe sua senha atual e que esta seja validada corretamente. Somente após a confirmação da senha, as alterações serão permitidas. Essa verificação evita modificações não autorizadas e assegura a integridade das informações cadastradas.

A.5.2 Sistema de Perfis e Permissões

Para garantir a segurança da API e possibilitar futuras melhorias no sistema, foi implementado um gerenciamento dinâmico de perfis e permissões. Dessa forma, a adição de novos perfis ou permissões pode ser feita de maneira prática e eficiente.

Para esse gerenciamento, foi integrada a biblioteca laravel-permission [12], da Spatie. Conforme descrito na documentação, esse pacote permite a criação e administração de perfis e permissões diretamente no banco de dados, vinculando-os ao modelo de usuário (SPATIE, 2025). Sua utilização é simples, bastando seguir as instruções iniciais na documentação [13].

Como a biblioteca é baseada em banco de dados, foi necessário configurar Seeders para facilitar a atualização dos perfis cadastrados. Isso permite, por exemplo, a adição de novas permissões sem complicações, garantindo flexibilidade e manutenção simplificada do sistema. Para o lançamento inicial, foi criado um único perfil, contendo todas as permissões disponíveis.

Como os perfis e permissões são registrados durante a execução das Seeders iniciais do sistema, foi criado um comando personalizado no *Laravel* para facilitar tanto a adição de novos perfis e permissões quanto a atualização de itens já registrados. Esse comando, conforme ilustrado na Figura 38, inicialmente executa a Seeder responsável por criar ou atualizar permissões e perfis, mas pode ser expandido para incluir novas ações, conforme necessário, para gerenciar essas configurações no sistema. No *Laravel*, os comandos podem ser executados diretamente no terminal através do Artisan, a Command-Line Interface (CLI) (Interface de Linha de Comando) do framework (LARAVEL, 2025c), proporcionando uma forma eficiente de automação e gestão de tarefas.

[12] https://spatie.be/docs/laravel-permission/v6/introduction
[13] https://spatie.be/docs/laravel-permission/v6/basic-usage/basic-usage

A.5.3 Exclusão de Solicitações de Manutenção Urbana

Com a implementação do cadastro de solicitações de manutenção urbana, tornou-se essencial desenvolver outra funcionalidade importante para a gestão desses registros: a exclusão de solicitações.

De acordo com os requisitos *RF 010* e *RF 016*, definidos no início do projeto, a exclusão só pode ser realizada caso duas condições sejam atendidas: a solicitação deve pertencer ao usuário que deseja excluí-la e não pode ter sido alterada por nenhum outro usuário, ou seja, não deve haver atualizações de sua situação. Se ambas as regras forem atendidas, a solicitação poderá ser excluída com sucesso.

Além da exclusão na API, essa funcionalidade também foi implementada no aplicativo. Ao excluir uma solicitação, todas as imagens vinculadas a ela, assim como seu histórico, são removidas do sistema. Isso evita o armazenamento de dados desnecessários e garante a integridade das informações.

A.6 Ciclo 6

Para o penúltimo ciclo de desenvolvimento, será descrita a implementação da funcionalidade de atualização da situação das solicitações de manutenção urbana, juntamente com a definição da delimitação da área de atuação do usuário.

A.6.1 Atualização de Situação da Solicitações de Manutenção Urbana

Após a implementação das funcionalidades de criação, listagem e exclusão de solicitações de manutenção urbana, o próximo passo foi desenvolver a atualização da situação, um recurso essencial no sistema. Essa funcionalidade permite que o usuário altere a situação de uma solicitação, que inicialmente está como “em aberto”, para “em progresso” e, posteriormente, para “concluída”, caso a demanda tenha sido devidamente resolvida.

No entanto, em alguns casos, a solicitação pode não ser resolvida ou, mesmo após o início da solução, pode ser interrompida por algum motivo. Nesses cenários, o fluxo de atualização de situação pode seguir um caminho diferente. O usuário que criou a solicitação pode, por exemplo, reverter uma solicitação da etapa “em progresso” para “em aberto”, em vez de avançar para a próxima etapa. Para padronizar e facilitar o controle da situação, foi criado um Enum, tanto no aplicativo quanto na API.

Independentemente da situação para a qual a solicitação está sendo alterada, é obrigatório anexar uma imagem capturada pela câmera, servindo como comprovação da mudança. Para realizar a atualização, o usuário precisa apenas selecionar a nova situação da solicitação e enviar a imagem correspondente. Assim que a alteração é concluída com sucesso, o sistema re-

gistra a modificação no histórico da solicitação, garantindo maior transparência e rastreabilidade para futuras consultas.

A.6.2 Delimitação de Área de Atuação do Usuário

O cidadão, como agente fundamental da sociedade e principal ator no sistema Ágora, assume um papel transformador em sua localidade e entorno. Para potencializar essa atuação, foi estabelecida uma delimitação espacial de 300 metros ao redor do usuário, definindo sua área de atuação dentro da plataforma. Ao acessar o aplicativo, o usuário visualiza, na tela do mapa, um círculo que representa essa área de 300 metros quadrados, centrada em sua localização atual.

Além disso, a listagem de solicitações, obtida por meio da API, é automaticamente filtrada para exibir apenas os itens que se encontram dentro desse raio de 300 metros, garantindo que o usuário tenha acesso às demandas mais relevantes e próximas de sua localização.

Para otimizar o desempenho e a experiência do usuário, a tabela de solicitações foi aprimorada com a inclusão da situação atual de cada item. Dessa forma, evita-se a necessidade de percorrer o histórico completo para identificar a última atualização, agilizando o carregamento das informações e proporcionando uma navegação mais eficiente.

A.7 Ciclo 7

Neste ciclo final da fase de desenvolvimento, foi implementada a última funcionalidade necessária para cumprir a lista de requisitos: a edição de solicitações. Além disso, foram realizados ajustes e melhorias significativas em aspectos como usabilidade, layout das telas e correções pontuais, visando garantir uma experiência mais intuitiva, ágil e eficiente para o usuário.

A.7.1 Edição de Solicitações de Manutenção Urbana

A edição das solicitações de manutenção urbana foi a última funcionalidade implementada no sistema, devido à sua complexidade. Para isso, foi criada uma nova tela no aplicativo, baseada na interface de cadastro, utilizando um formulário dividido em etapas.

Na primeira etapa, o usuário pode editar informações como título, descrição e categoria. Já na segunda, é possível gerenciar as imagens da solicitação, permitindo a exclusão das fotos anteriores e a adição de novas, seguindo a mesma regra do cadastro: mínimo de uma e máximo de três imagens.

Cada atualização realizada gera automaticamente um registro no histórico da solicitação, garantindo rastreabilidade e transparência no acompanhamento das alterações.

A.7.2 Melhorias e Ajustes Finais do Aplicativo e API

Após a finalização do desenvolvimento das funcionalidades básicas do sistema e a realização dos primeiros testes com o aplicativo instalado em um dispositivo móvel, foram identificadas algumas melhorias a serem implementadas. Entre elas, destacam-se ajustes no layout, como a remoção dos cabeçalhos das telas e do botão de voltar nativo, que são padrões para a navegação do Expo. Para resolver isso, o componente padrão foi substituído, os cabeçalhos das telas foram removidos e um novo componente personalizado foi criado para a navegação entre as telas.

Além disso, foram feitos ajustes no botão de captura de imagens, garantindo que ele fosse bloqueado após a captura de uma nova foto e, também, refinamentos em outras funcionalidades. Por fim, a tela de login foi redesenhada, visando proporcionar uma melhor experiência ao usuário e reforçar a identidade visual do sistema Ágora no momento do acesso.

```tsx
import {
  createContext,
  PropsWithChildren,
  useContext,
  useEffect,
  useState,
} from 'react';
import * as Location from 'expo-location';
import { errorToast } from '@/utils/use-toast';

interface LocationCoordinateProps {
  latitude: number;
  longitude: number;
  setLatitude: (latitude: number) => void;
  setLongitude: (longitude: number) => void;
}

const LocationCoordinatesContext =
  createContext<LocationCoordinateProps | null>(null);

export function LocationCoordinatesContextProvider({ children }: PropsWithChildren) {
  const [latitude, setLatitude] = useState(0);
  const [longitude, setLongitude] = useState(0);

  useEffect(() => {
    const getLocation = async () => {
      try {
        const permission = await Location.requestForegroundPermissionsAsync();

        if (permission) {
          const location = await Location.getCurrentPositionAsync({});

          setLatitude(location.coords.latitude);
          setLongitude(location.coords.longitude);
        } else {
          errorToast({
            title:
'Permissão de acesso de localização em tempo real negada! Esta ação é necessária, para criar novas
solicitações!'
          });
        }
      } catch (error) {
        throw Error;
      }
    };

    getLocation();
  }, []);

  return (
    <LocationCoordinatesContext.Provider
      value={{ latitude, longitude, setLatitude, setLongitude }}
    >
      {children}
    </LocationCoordinatesContext.Provider>
  );
}

export function useLocationCoordinates() {
  const context = useContext(LocationCoordinatesContext);

  if (!context)
    throw new Error(
      'useLocationCoordenates must be used inside LocationCoordenatesContext',
    );

  return context;
}
```

Figura 29 – Contexto para controle e acesso de localização do dispositivo móvel via GPS.

Fonte: Autoria própria.

```php
<?php

namespace App\Http\Api\Request\Client\Solicitation;

use Illuminate\Contracts\Validation\ValidationRule;
use Illuminate\Foundation\Http\FormRequest;

class SolicitationRequest extends FormRequest
{
    /**
     * Determine if the user is authorized to make this request.
     */
    public function authorize(): bool
    {
        return true;
    }

    /**
     * Get the validation rules that apply to the request.
     *
     * @return array<string, ValidationRule|array|string>
     */
    public function rules(): array
    {
        return [
            'title' => 'required|string|min:5|max:255',
            'description' => 'required|string|min:5|max:1000',
            'latitudeCoordinates' => 'required|string',
            'longitudeCoordinates' => 'required|string',
            'solicitationCategoryId' => 'required|exists:solicitation_categories,id',
        ];
    }
}
```

Figura 30 – Request para validação de dados de entrada para cadastro/edição de solicitações de manutenção urbana.

Fonte: Autoria própria.

```php
<?php

namespace App\Domains\Solicitation\Dtos;

use Spatie\LaravelData\Attributes\DataCollectionOf;
use Spatie\LaravelData\Attributes\Validation\Exists;
use Spatie\LaravelData\Attributes\Validation\IntegerType;
use Spatie\LaravelData\Attributes\Validation\Max;
use Spatie\LaravelData\Attributes\Validation\Min;
use Spatie\LaravelData\Attributes\Validation\Nullable;
use Spatie\LaravelData\Attributes\Validation\Required;
use Spatie\LaravelData\Attributes\Validation\StringType;
use Spatie\LaravelData\Data;

class SolicitationData extends Data
{
    public function __construct(
        #[Required, StringType, Min(5), Max(255)]
        public string $title,

        #[Required, StringType, Min(5), Max(1000)]
        public string $description,

        #[Required, StringType, Min(1), Max(15)]
        public string $latitudeCoordinates,

        #[Required, StringType, Min(1), Max(15)]
        public string $longitudeCoordinates,

        #[Required, IntegerType, Exists('solicitation_categories', 'id')]
        public int $solicitationCategoryId,

        #[Nullable, IntegerType]
        public ?int $likesCount,

        #[Nullable, DataCollectionOf(UserSolicitationData::class)]
        public ?UserSolicitationData $userSolicitationData,

        #[Nullable, StringType]
        public ?string $currentStatus,
    ) {}
}
```

Figura 31 – DTO para dados de solicitação de manutenção urbana.

Fonte: Autoria própria.

```php
<?php

namespace App\Domains\Solicitation\Actions\Solicitation;

use App\Domains\Solicitation\Dtos\SolicitationData;
use App\Domains\Solicitation\Models\Solicitation;

class CreateSolicitationAction
{
    public function execute(SolicitationData $data): Solicitation
    {
        $data = array_keys_as($data->toArray(), [
            'latitudeCoordinates' => 'latitude_coordinates',
            'longitudeCoordinates' => 'longitude_coordinates',
            'solicitationCategoryId' => 'solicitation_category_id',
            'currentStatus' => 'current_status',
        ]);

        return app(Solicitation::class)
            ->create($data);
    }
}
```

Figura 32 – Action para cadastro de nova solicitação de manutenção urbana.

Fonte: Autoria própria.

```php
<?php

namespace Tests\Unit\Domains\Solicitation\Actions\Solicitation;

use App\Domains\Solicitation\Actions\Solicitation\CreateSolicitationAction;
use App\Domains\Solicitation\Dtos\SolicitationData;
use App\Domains\Solicitation\Models\Solicitation;
use Tests\Cases\TestCaseUnit;

class CreateSolicitationActionTest extends TestCaseUnit
{
    public function test_should_create_solicitation()
    {
        $data = SolicitationData::from([
            'title' => 'title',
            'description' => 'description',
            'latitudeCoordinates' => 1,
            'longitudeCoordinates' => 2,
            'solicitationCategoryId' => 3,
        ]);

        $this->mock(Solicitation::class)
            ->expects('create')
            ->andReturns(new Solicitation);

        (new CreateSolicitationAction)->execute($data);
    }
}
```

Figura 33 – Teste unitário desenvolvido para testar execução da Action para cadastro de nova solicitação de manutenção urbana.

Fonte: Autoria própria.

```typescript
export const createSolicitation = async
 (data: SolicitationDataInterface): Promise<SolicitationResponseInterface> => {
  try {
    const response = await axiosInstance()
      .post(apiRoutes.mySolicitations.index, data);

    return response.data.data;

  } catch (error: any) {
    throw error;
  }
};
```

Figura 34 – Service para cadastro de novas solicitações de manutenção urbanas, utilizado no aplicativo.

Fonte: Autoria própria.

```
import * as FileSystem from 'expo-file-system';
import AsyncStorage from '@react-native-async-storage/async-storage';

interface FileSystemUploadAsyncProps {
  url: string;
  imageUri: string;
}

export const uploadAsyncImageUri = async
 ({ url, imageUri }: FileSystemUploadAsyncProps) => {
  const token = await AsyncStorage.getItem('token');

  await FileSystem
    .uploadAsync(url, imageUri, {
      httpMethod: 'POST',
      uploadType: FileSystem.FileSystemUploadType.BINARY_CONTENT,
      headers: {
        'Authorization': `Bearer ${token}`,
      },
    });
};
```

Figura 35 – Classe base de upload de imagens utilizada na aplicação.

Fonte: Autoria própria.

```php
<?php

namespace App\Domains\Solicitation\Models;

use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Database\Eloquent\Model;
use Spatie\MediaLibrary\HasMedia;
use Spatie\MediaLibrary\InteractsWithMedia;

class Solicitation extends Model implements HasMedia
{
    use HasFactory, InteractsWithMedia;

    protected $table = 'solicitations';

    protected $fillable = [
        'solicitation_category_id',
        'title',
        'description',
        'latitude_coordinates',
        'longitude_coordinates',
        'likes_count',
        'current_status',
    ];

    public function registerMediaCollections(): void
    {
        $this->addMediaCollection('coverImage')
            ->singleFile();

        $this->addMediaCollection('images');
    }
}
```

Figura 36 – Exemplo de modelo configurado para poder usar imagens com a biblioteca laravel-medialibrary.

Fonte: Autoria própria.

```php
$solicitation->addMedia($tempImageFile)
                    ->toMediaCollection('coverImage');
```

Figura 37 – Exemplo de vinculação de imagem a modelo utilizando o pacote laravel-medialibrary.

Fonte: Autoria própria.

```php
<?php

namespace App\Console\Commands;

use App\Domains\Account\Models\User;
use Database\Seeders\RolesAndPermissionsSeeder;
use Illuminate\Console\Command;

class UpgradeRolesAndPermissionsCommand extends Command
{
    /**
     * The name and signature of the console command.
     *
     * @var string
     */
    protected $signature = 'upgrade-roles-and-permissions';

    /**
     * The console command description.
     *
     * @var string
     */
    protected $description = 'Upgrade API roles and permissions according to RolesAndPermissions class.';

    /**
     * Execute the console command.
     */
    public function handle(): void
    {
        $this->info('Running RolesAndPermissionsSeeder...');
        $this->call(RolesAndPermissionsSeeder::class);
        $this->info('RolesAndPermissionsSeeder completed!');
    }
}
```

Figura 38 – Comando personalizado para atualização de perfis e permissões através do terminal da aplicação com o uso do Artisan.

Fonte: Autoria própria.

APÊNDICE B – Modelagem do Banco de Dados do Sistema

Neste apêndice será apresentada a modelagem final do banco de dados, que foi primeiramente discutida e apresentada na seção 5.2, Modelo Entidade Relacionamento (MER), e que sofreu modificações durante a implementação do sistema para atender melhor às necessidades do projeto.

B.0.1 Tabelas de Endereço

Para armazenar o endereço do usuário, foi criado um grupo de tabelas: *address_states*, *address_cities* e *addresses*. Cada uma delas possui uma função específica: a primeira armazena os estados, a segunda os municípios e a terceira contém os detalhes do endereço do usuário, como o CEP. A tabela *addresses* se relaciona diretamente com a tabela de usuários (*users*) para vincular cada endereço a um usuário específico.

Os estados são cadastrados automaticamente no banco de dados por meio de Seeders durante a configuração inicial da aplicação. Essa estrutura foi projetada para evitar a redundância de informações, garantindo que nomes de cidades e estados não sejam armazenados repetidamente. Essas primeiras tabelas estão demonstradas na Figura 39.

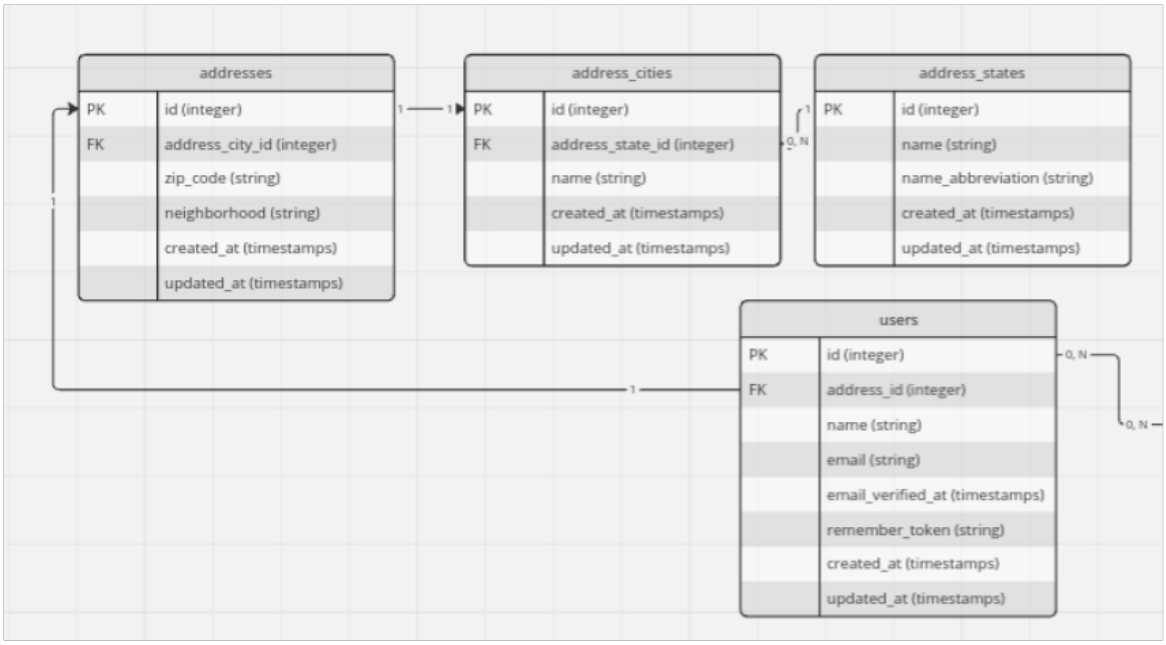


Figura 39 – Diagrama de tabelas que correspondem ao armazenamento de dados de endereço dos usuários do sistema.

Fonte: Autoria própria.

B.0.2 Tabelas de Usuários e Solicitações

A tabela responsável por armazenar os usuários é a *users*. E não sofreu modificações desde sua primeira versão. Ela se relaciona com a tabela *addresses* por meio da chave estran-

geira *address_id*, garantindo que cada usuário possa ter apenas um endereço. Nesta mesma tabela, são armazenadas as credenciais utilizadas para o login, sendo que a senha é salva de forma criptografada para proteger os dados.

A tabela users_solicitations atua como uma intermediária entre as tabelas de usuários e solicitações. Ela registra o histórico de atualizações das solicitações, os reforços recebidos de outros usuários e a situação da solicitação no momento de cada ação. Essa estrutura unifica as antigas tabelas *likes* e *solicitations_histories*, consolidando os dados em um único local para armazenar tanto o histórico de alterações quanto os reforços recebidos.

Já a tabela *solicitations* armazena os dados de cada solicitação, incluindo título, descrição, categoria, localização geográfica (em coordenadas do local onde foi criada via aplicativo), total de reforços recebidos e sua situação atual. Este último campo foi adicionado para otimizar o desempenho das consultas, evitando a necessidade de percorrer todo o histórico de uma solicitação para determinar sua etapa atual.

As categorias das solicitações são armazenadas na tabela *solicitations_categories* nesta tabela foi adicionado um campo para armazenar a descrição para as categorias. Essa estrutura também facilita futuras integrações da aplicação com um painel administrativo, possibilitando o cadastro de novas categorias de forma mais ágil e eficiente.

A tabela *media* é criada automaticamente com a instalação da biblioteca laravel-medialibrary, desenvolvida pela Spatie. Sua função é armazenar e associar as imagens das solicitações, sejam elas enviadas no momento da criação, edição ou como evidência para atualização da situação. Além disso, essa tabela estabelece a relação entre os arquivos e os modelos do *Laravel*, garantindo um gerenciamento eficiente da mídia no sistema. Com sua implementação, a tabela *solicitation_images*, prevista no planejamento inicial do banco de dados, tornou-se desnecessária.

Além dessas tabelas, o banco de dados também inclui as tabelas padrão do *Laravel* quando o Sanctum é utilizado como método de autenticação. Entre elas, destacam-se *personal_access_tokens* e *password_reset_tokens*, responsáveis pelo gerenciamento de tokens de acesso e redefinição de senha, respectivamente.

Essas tabelas, juntamente com as relacionadas a usuários e solicitações, estão representadas na Figura 40.

B.0.3 Tabelas de Perfis e Permissões

Como mencionado na subseção A.5.2, foi implementado um sistema de gerenciamento de perfis e permissões no sistema, através da biblioteca laravel-permissions. Ao adicionar esta biblioteca, automaticamente são criadas as migrações para o banco de dados para criar de forma automática as tabelas de perfis e permissões.

A tabela *roles* armazena os perfis criados, enquanto a tabela *permissions* gerencia as permissões que podem ser atribuídas aos usuários no sistema. A relação entre perfis e permis-

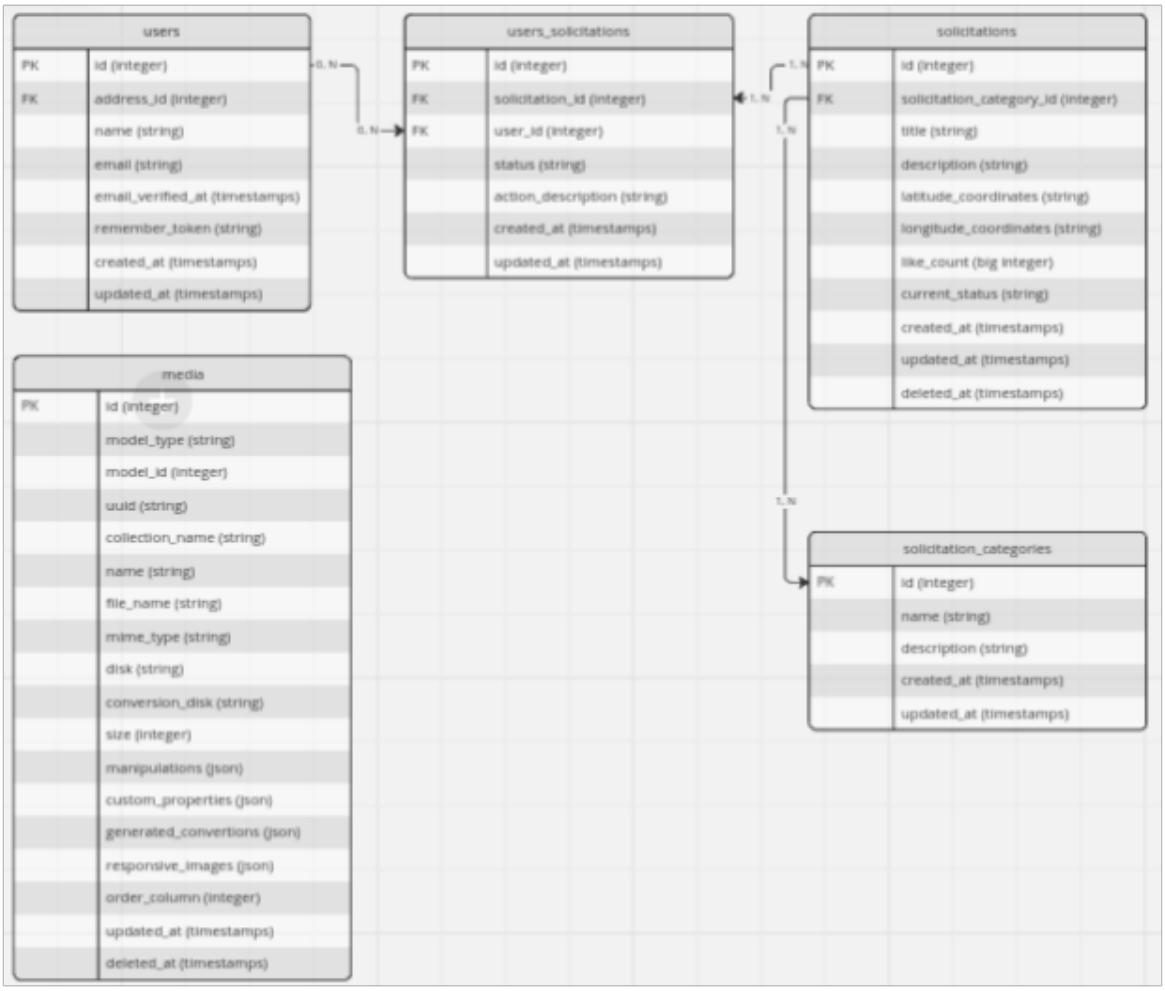


Figura 40 – Diagrama de tabelas que correspondem ao armazenamento de dados de usuários e solicitações.

Fonte: Autoria própria.

sões é feita pela tabela *role_has_permissions*, que vincula cada perfil às permissões correspondentes.

Além disso, as tabelas *model_has_roles* e *model_has_permissions* estabelecem a ligação entre os modelos e os perfis ou permissões atribuídos a eles. Ou seja, a primeira relaciona os modelos aos perfis, enquanto a segunda faz o vínculo entre os modelos e suas permissões específicas. Essas tabelas estão representadas na Figura 41.

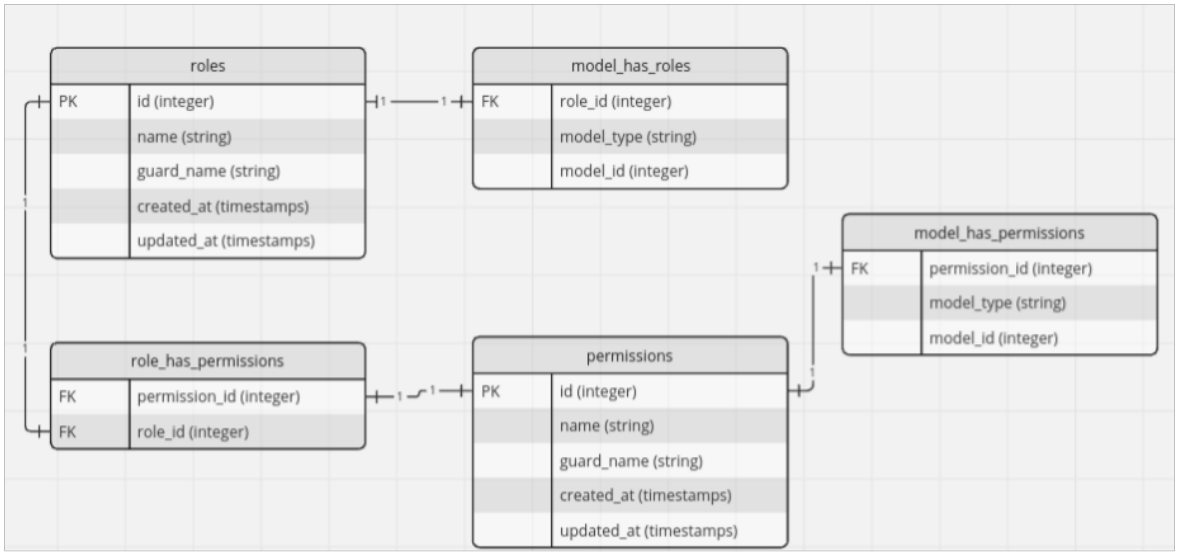


Figura 41 – Diagrama de tabelas que correspondem ao armazenamento de perfis e permissões do sistema.

Fonte: Autoria própria.