Auchei miaumigo: Um sistema *web* para adoção e localização de animais perdidos

Yan Valadares. T. Cardos [1], Leandro C. Ledel[2],

Análise e Desenvolvimento de Sistemas - Instituto Federal de Educação, Ciência e Tecnologia de São Paulo - Campus Hortolândia
Hortolândia – SP – Brasil

y.cardoso@aluno.ifsp.edu.br, ledel@ifsp.edu.br

Abstract. *The number of animals in vulnerable situations has gradually increased in recent years, as has the population of animals in shelters and NGOs. Given this scenario, this article presents the development of Auchei Miaumigo, a web-based system focused on pet adoption and lost animal tracking. The software aims to streamline the adoption process by connecting NGOs and adopters while also providing a platform for searching and locating missing animals. The project development was structured into three main stages: frontend, involving user interface prototyping and code implementation; backend, responsible for database modeling and creation, business rule definition, and REST API development; and testing, encompassing unit, integration, and end-to-end tests to ensure the application's quality and reliability. Through this approach, Auchei Miaumigo seeks to optimize and encourage both the adoption process and the tracking of lost animals, contributing to an increase in successful adoptions and reunions.*

Resumo. *O número de animais em situação de vulnerabilidade tem aumentado gradativamente nos últimos anos, assim como a população de animais em abrigos e ONGs. Diante desse cenário, este artigo apresenta o desenvolvimento do Auchei Miaumigo, um sistema web voltado para adoção e localização de animais perdidos. O software tem como objetivo facilitar o processo de adoção, conectando ONGs e adotantes, além de oferecer uma plataforma para busca e localização de animais desaparecidos. O desenvolvimento do projeto foi estruturado em etapas: frontend, englobando a prototipação da interface de usuário e a implementação do código; backend, responsável pela modelagem e criação do banco de dados, definição das regras de negócio e desenvolvimento da API REST; e testes, abrangendo testes unitários, de integração e end-to-end para garantir a qualidade e confiabilidade da aplicação. Com essa abordagem, o Auchei Miaumigo busca otimizar e incentivar tanto o processo de adoção quanto a localização de animais perdidos, contribuindo para o aumento do número de adoções e reencontros bem-sucedidos.*

1. Introdução

Nos últimos anos, o número de animais em situação de vulnerabilidade vem crescendo de forma alarmante (Puente, 2022), contribuindo para o aumento da população de animais em abrigos e ONGs, bem como para um maior índice de abandono, resultando no aumento da quantidade de animais de rua. Esse preocupante aumento resulta em um problema de saúde pública, além de um desequilíbrio ambiental (Severino, 2023), o que requer a implementação de medidas para mitigar esse problema. As ONGs desempenham um papel crucial na sociedade ao adotarem medidas para reduzir a presença de animais nas ruas e facilitar a devolução deles aos seus lares. Assim, essas organizações são fundamentais na sociedade atual, devido aos benefícios e à significativa função social que desempenham, tornando essencial a disponibilização de recursos e suporte adequados para essas instituições. No entanto, atualmente as ONGs enfrentam um desafio

significativo: resgatam mais animais do que são efetivamente adotados (Pastori, 2015), resultando em um sério problema social e ambiental.

Outro aspecto importante é a dificuldade em localizar animais desaparecidos, levando muitas vezes à sua permanência nas ruas. Além dos problemas mencionados anteriormente, o desaparecimento de um animal pode causar danos emocionais ao tutor e sua família. Com a escassez de recursos auxiliares para localizar o animal desaparecido, a probabilidade de encontrá-lo diminui consideravelmente.

A tecnologia tem possibilitado a visibilidade de diversos problemas sociais que antes passavam despercebidos, trazendo à tona questões importantes para o nosso cotidiano, por meio de uma maior facilidade e agilidade na troca de informação entre pessoas. Ao disponibilizar um sistema *web* que facilite o acesso a essas informações, mais pessoas podem tomar conhecimento da situação dos animais abrigados, assim se solidarizando com os mesmos, visando um aumento no número de adoções e na contribuição para essa importante causa, além de facilitar a localização de animais abandonados.

Diante dos fatores apresentados, este trabalho tem como objetivo o desenvolvimento de uma aplicação *web* para aprimorar o processo de adoção e localização de animais abandonados. O sistema permitirá que organizações de proteção animal publiquem informações sobre os animais sob seus cuidados, facilitando o encontro de adotantes interessados e a iniciação do processo de adoção. Além disso, a aplicação oferecerá recursos para ajudar na localização de animais desaparecidos, por meio de publicações feitas por seus tutores, o que agiliza a busca e aumenta a chance de recuperação. Essas funcionalidades visam enfrentar os desafios na proteção animal com intuito de preservar e promover os direitos dos animais estabelecidos (UNESCO, 1978), contribuindo para a redução de animais abandonados nas ruas, promovendo a adoção responsável.

2. Fundamentação teórica

Nesta seção serão abordadas as questões fundamentais para o desenvolvimento da aplicação, descrevendo e definindo os principais conceitos e ferramentas utilizadas.

2.1. Desenvolvimento *web*

O desenvolvimento *web* é utilizado para disponibilizar uma aplicação para ser acessada por meio da *internet*, assim o sistema pode ser acessado por dispositivos que tenham acesso a mesma. Dessa forma, uma aplicação *web* tende a ser mais disponível aos usuários, facilitando seu uso. Consiste em dois principais pontos, sendo eles: *frontend* (interface visual), acessada pelo cliente por meio de um navegador instalado em sua máquina (Pacheco, 2023) e *backend* (lógica e banco de dados).

2.2. *TypeScript*

TypeScript é uma linguagem de programação fortemente tipada que se baseia em *JavaScript*, oferecendo melhores ferramentas em qualquer escala. Assim, a linguagem proporciona tanto o dinamismo promovido pelo *JavaScript* nativo quanto uma maior confiabilidade no código escrito, manutenção de forma mais simples e prevenção de comportamentos inesperados no

software. Por esses motivos, o *TypeScript* foi escolhido para estar presente tanto no *backend* quanto no *frontend* (*Typescript, 2024*).

2.3. *Node.js*

Ambiente de execução de *JavaScript* que permite a criação de servidores (*Node.js,* 2024). Por meio da utilização do motor v8, desenvolvido e utilizado pela *Google* para interpretar *JavaScript* nos navegadores com base *Chrome* (v8, 2024), o *Node.js* é capaz de interpretar a linguagem *JavaScript* no lado do servidor, permitindo a utilização da linguagem não só no lado do cliente, mas também no lado do servidor.

2.4. *Frontend*

O *frontend* pode ser descrito como a parte interativa, a qual o usuário pode interagir com a aplicação. Esta parte da aplicação é importante pois é onde estão disponibilizadas as funcionalidades do sistema de forma amigável ao usuário, facilitando com o mesmo localize o que está procurando e realize a tarefa de forma simples.

2.4.1. *React*

Biblioteca utilizada para criar interfaces de usuário utilizando componentes reutilizáveis, utilizando o *Node.js* para executar o *JavaScript* fora do contexto do navegador (*React, 2024*). Cada um desses componentes é uma parte independente da interface, com sua própria lógica e estado, facilitando assim a construção de uma interface modular, permitindo uma manutenção facilitada e maior organização do código. Além disso, o *React* utiliza uma DOM (*Document Object Model*) virtual, com essa técnica, é possível comparar a mudança de estados e renderizar apenas os componentes modificados, o que melhora o desempenho da aplicação.

2.4.2 *Next.js*

O *Next.js* é um *framework* que utiliza o *React* para a criação de aplicações *web* (*Next.js, 2024*). Uma das características do *Next.js* é a renderização das páginas no lado do servidor, ou seja, as páginas são pré-processadas no lado do servidor antes de serem enviadas para o cliente, reduzindo a carga de processamento pelo lado do cliente, aumentando a performance da aplicação e permitindo suporte à conteúdos dinâmicos de forma mais eficiente.

2.5. *Backend*

O *backend* é a parte do sistema responsável pelo processamento dos dados recebidos e que serão disponibilizados para o *frontend*. Nessa parte da aplicação estão localizadas as regras de negócio, a lógica da aplicação, a comunicação com o banco e o processamento desses dados.

2.5.1. *Fastify*

O *fastify* é um *framework* que utiliza o *Node.js* para a criação de servidores. Como o próprio nome pode indicar, o *fastify* visa uma maior velocidade no processamento de requisições e no envio de respostas, além de ser totalmente extensível por meio de plugins (*Fastify, 2024*). Outro benefício do uso desse *framework* é o alto suporte a linguagem *TypeScript*, linguagem que será utilizada em todo projeto.

2.6. Banco de dados

Um banco de dados é uma coleção organizada de informações estruturadas, armazenadas eletronicamente em um sistema de computador. Ele é controlado por um Sistema Gerenciador de Banco de Dados (SGBD), que facilita o acesso, gerenciamento e manipulação dos dados (*Oracle, 2020*). O banco de dados é organizado em tabelas formadas por linhas, que são responsáveis identificar os registros feitos no banco, e colunas, os atributos de cada registro realizado.

2.6.1. *PostgreSQL*

O *PostgreSQL* é um sistema gerenciador de banco de dados (SGBD) relacional de código aberto, com foco em robustez e confiabilidade (*Postgres, 2024*). Por meio da linguagem SQL (*Structured Query Language*) a escrita de consultas e novos registros no banco de dados pode ser realizada, além de permitir também a exclusão dos registros.

2.6.2. ORM

ORM é uma sigla para *Object-Relational Mapping*, sendo essa uma técnica de programação que permite a interação com bancos de dados relacionais usando conceitos da programação orientada a objetos. Ao invés de consultas SQL serem escritas para interação com o banco de dados, são utilizadas classes e objetos que são responsáveis por essa interação, permitindo assim a criação de uma camada de abstração entre a aplicação e o banco de dados e também aumentando a portabilidade do sistema, visto que são praticamente independentes a SGBDs.

2.6.3. *Prisma* ORM

Segundo a documentação da ferramenta, o *Prisma* ORM possui integração com o *Node.js* e o *Typescript*, permitindo uma interação simples entre o servidor e o banco de dados. Essa ferramenta permite definir modelos, que representam a tabela dentro do banco de dados, gerenciar migrações e realizar consultas de forma segura, simples e tipada, sem a necessidade da escrita de código SQL (Prisma, 2024).

2.7. API

API (*Application Programming Interface*) são mecanismos que permitem que dois componentes de *software* se comuniquem usando um conjunto de definições e protocolos (Amazon, 2023). Essa comunicação é feita por meio de requisições realizadas por um cliente para um servidor, responsável por processar essa requisição e responder essa solicitação.

2.7.1 API REST

API REST são um tipo de API que permite a comunicação entre cliente e servidor por meio do protocolo HTTP, de forma semelhante a páginas *web* (Amazon, 2023). Algumas características de APIs REST são a falta de estado, ou seja, o servidor não mantém informações sobre as solicitações anteriores do cliente, além da resposta do servidor ser geralmente no formato JSON, sendo esse um formato baseado em texto padrão para representar dados estruturados com base na sintaxe do objeto *JavaScript* (MDN, 2024).

2.7.2. API IBGE

A API do IBGE (Instituto Brasileiro de Geografia e Estatística) fornece dados referentes aos países e às divisões político-administrativas do Brasil bem como meso e microrregiões,

institucionalizadas pela aprovação da presidência do IBGE (IBGE, 2024). Dessa forma, a API fornece informações como os Estados brasileiros, além de cada cidade que esse Estado possui.

2.8. Testes

Testes de *software* podem ser entendidos como qualquer procedimento que determina se o sistema consegue atingir as especificações as quais foram definidas para ele (Félix, 2016). Visando garantir as expectativas propostas na criação do sistema, testes tanto para o *frontend* quanto para o *backend* foram desenvolvidos. Ao todo, três tipos de testes foram utilizados, sendo eles os testes unitários, testes de integração e testes *end-to-end*. Segundo a pirâmide de testes, proposta inicialmente por Mike Cohn, presente na Figura 1, um *software* deve possuir testes unitários em abundância, alguns testes de integração e poucos testes *end-to-end*. Isso se dá ao fato de, quanto mais acoplados, mais recursos e tempo os testes requerem.

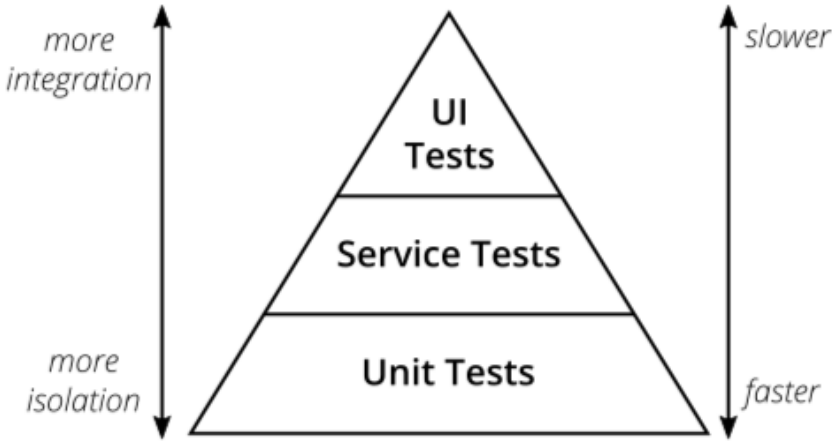

Figura 1. Pirâmide de testes. Fonte: www.martinfowler.com/articles/practical-test-pyramid.

3. Trabalhos correlatos

Nesta seção, os trabalhos relacionados ao sistema a ser desenvolvido serão abordados. Conforme a pesquisa realizada na revisão bibliográfica, alguns trabalhos foram selecionados para análise conforme pontos específicos apresentados. Dentre essas aplicações escolhidas, estão o AdotePetz, plataforma de adoção da empresa Petz; o PetAdote, plataforma que visa auxiliar na adoção e divulgação de animais; e o LocPet, aplicativo para localização de animais perdidos.

3.1. AdotePetz

O AdotePetz é uma plataforma que tem o intuito de conectar as pessoas interessadas em adotar com os animais em ONGs e abrigos parceiros da empresa. O *site* apresenta uma interface intuitiva, permitindo que os usuários filtrem os animais disponíveis por meio da localização, espécie, porte e outras características. O *site* também apresenta uma descrição detalhada dos animais disponíveis para adoção, como fotos, idade, descrição do animal, entre outros.

3.2. PetAdote

O PetAdote é uma aplicação *web* desenvolvida para facilitar a adoção e divulgação de animais domésticos abandonados, conectando diretamente protetores e potenciais adotantes. O sistema visa simplificar processo de adoção, reduzindo burocracias e aumentando a visibilidade dos animais, aumentando a chance de encontrarem um lar adequado. A interface do sistema é intuitiva, atendendo as necessidades dos usuários e provendo uma experiência agradável.

3.3. LoctPet

O LocPet é um aplicativo móvel projetado para facilitar a localização de animais perdidos ou abandonados, utilizando geolocalização fornecida pelos próprios usuários. A plataforma permite que pessoas comuniquem avistamentos de animais nas ruas ou solicitem ajuda para encontrar seus *pets* desaparecidos. Além de contribuir para a redução de animais em situação de rua, o LocPet promove a adoção responsável, divulga feiras de adoção, e campanhas de vacinação e castração, sensibilizando a população sobre a importância de cuidar e proteger os animais.

3.4. Comparação entre sistemas

Características	AdotePetz	PetAdote	LocPet	Auchei Miaumigo
Plataforma	Web	Web	Mobile	Web
Facilita o processo de adoção	Sim	Sim	Não	Sim
Auxilia na localização de animais perdidos	Não	Não	Sim	Sim
Comunicação com a ORG	Sim	Sim	Não	Sim
Filtro de busca	Sim	Sim	Não	Sim

Figura 2. Tabela de comparação entre sistemas. Autoria própria.

4. Metodologia

Como forma de organizar as ideias e promover um desenvolvimento mais objetivo, algumas etapas, as quais podem ser encontradas nas subseções seguintes, foram utilizadas para nortear o desenvolvimento do sistema.

4.1. Revisão bibliográfica

Etapa que visa a análise de diferentes pontos de vista, além de proporcionar estatísticas e uma visão mais abrangente sobre o tema, o que influenciará na formulação de proposta para uma solução mais precisa dos problemas encontrados.

4.2. Levantamento de requisitos

Por meio da análise de alguns *sites* de adoção de animais, como por exemplo o AdotePetz, *website* que permite a visualização de uma gama de animais em diferentes localidades do Brasil, os requisitos do sistema e suas funcionalidades puderam ser definidas.

4.3. Criação de diagramas

Após o levantamento de requisitos, os diagramas de casos de uso e de classes foram criados visando permitir uma visão mais detalhada sobre as especificações do sistema, além de possibilitar uma análise mais abrangente sobre a estrutura da aplicação.

4.4. Modelagem de banco de dados

Nessa etapa, o modelo conceitual do banco de dados foi elaborado com o objetivo de fornecer uma visão abrangente do banco de dados utilizado no sistema, adaptando-o aos requisitos e funcionalidades específicas. Essa fase do projeto também facilita a visualização global do banco de dados, o que contribui para o processo de normalização. Segundo a *Microsoft*, a normalização é o processo de organização dos dados em um banco de dados por meio de regras projetadas para proteger a integridade dos dados e aumentar a flexibilidade do banco, eliminando redundâncias e dependências inconsistentes.

4.5. Desenvolvimento *backend*

Etapa em que foram definidas as tecnologias, ferramentas e a arquitetura do *backend* do sistema, além da linguagem de programação. Além disso, uma arquitetura baseada na *Use-case Architecture* foi escolhida por seu foco no domínio da aplicação, permitindo que as funcionalidades do sistema sejam modeladas de maneira a refletir as regras de negócio e necessidades do projeto. Essa escolha é importante, pois o domínio é o ponto central do sistema, e o uso dessa arquitetura garante que as decisões de *design* estejam alinhadas com os objetivos e desafios do negócio. Uma definição mais detalhada da arquitetura pode ser encontrada no apêndice 1.

4.6. Desenvolvimento *frontend*

Assim como na etapa do desenvolvimento do *backend*, as ferramentas e as tecnologias foram definidas. Além disso, a criação de protótipos das telas foram produzidas no *Figma*, ferramenta que permite a criação de protótipos das telas. A partir das telas criadas, é possível definir um padrão dos componentes da interface, permitindo que toda a aplicação *frontend* siga padrões, como, por exemplo, os espaçamentos verticais entre componentes ou o tamanho e o arredondamento de botões.

4.7. Testes

Os testes unitários são uma forma de garantir a integridade da aplicação, proporcionando um *feedback* em tempo real após a escrita de diferentes partes do código. Estes testes permitem uma alta cobertura das linhas de código e a detecção precoce de erros, conforme sugerido por Hunt e Thomas (2003). Além disso, os testes *end-to-end* tem como propósito assegurar que todas as funcionalidades do sistema funcionem corretamente em conjunto, simulando o fluxo completo da aplicação.

5. Desenvolvimento

Nesta seção serão abordados os principais pontos do desenvolvimento do sistema.

5.1. Levantamento de requisitos

Por meio da revisão dos trabalhos presentes na seção de Trabalhos Correlatos, os requisitos funcionais e não funcionais do sistema puderam ser definidos. Requisitos funcionais, como definidos por Sommerville, são as declarações dos serviços que o sistema deve oferecer, além da forma como a aplicação deve se comportar devido a determinadas situações. Por outro lado, os requisitos não funcionais são as restrições e/ou funções oferecidas pelo sistema como um todo.

Alguns dos requisitos funcionais do sistema são: Cadastrar animal perdido; Procurar animal; Procurar ONG; e Requisitar adoção. Todos os requisitos funcionais podem ser encontrados na Figura 3, que representa o diagrama de casos de uso do sistema.

Já os requisitos não funcionais são: segurança, obtida por meio da validação do usuário durante a interação com as funcionalidades do sistema; integridade de dados, realizada por meio de validações dos dados durante todo o processo de criação das entidades; e desempenho, é adquirido por meio da utilização de *frameworks* e padrões de códigos utilizados na aplicação.

5.2. Criação de diagramas

Por meio da análise dos requisitos levantados no tópico anterior, o diagrama de casos de uso e o diagrama de classes foram criados para uma permitir uma visão geral da aplicação, assim como os requisitos e as funcionalidades do sistema.

O diagrama de casos de uso, representado na Figura 3, pode ser definido como um "índice gráfico" que representa os casos de uso e os atores de um sistema, além de ilustrar a interação dos atores com esses casos (Valente, 2023). Por meio deste diagrama, é possível compreender as funcionalidades do sistema e como cada tipo de usuário interage com a aplicação. No diagrama, conseguimos identificar três atores, sendo eles o Tutor, o Administrador de ONG e o Usuário não logado. Além disso, é possível identificar as funcionalidades disponíveis para cada ator, quais funções são individuais de cada tipo de usuário e quais são compartilhadas entre eles.

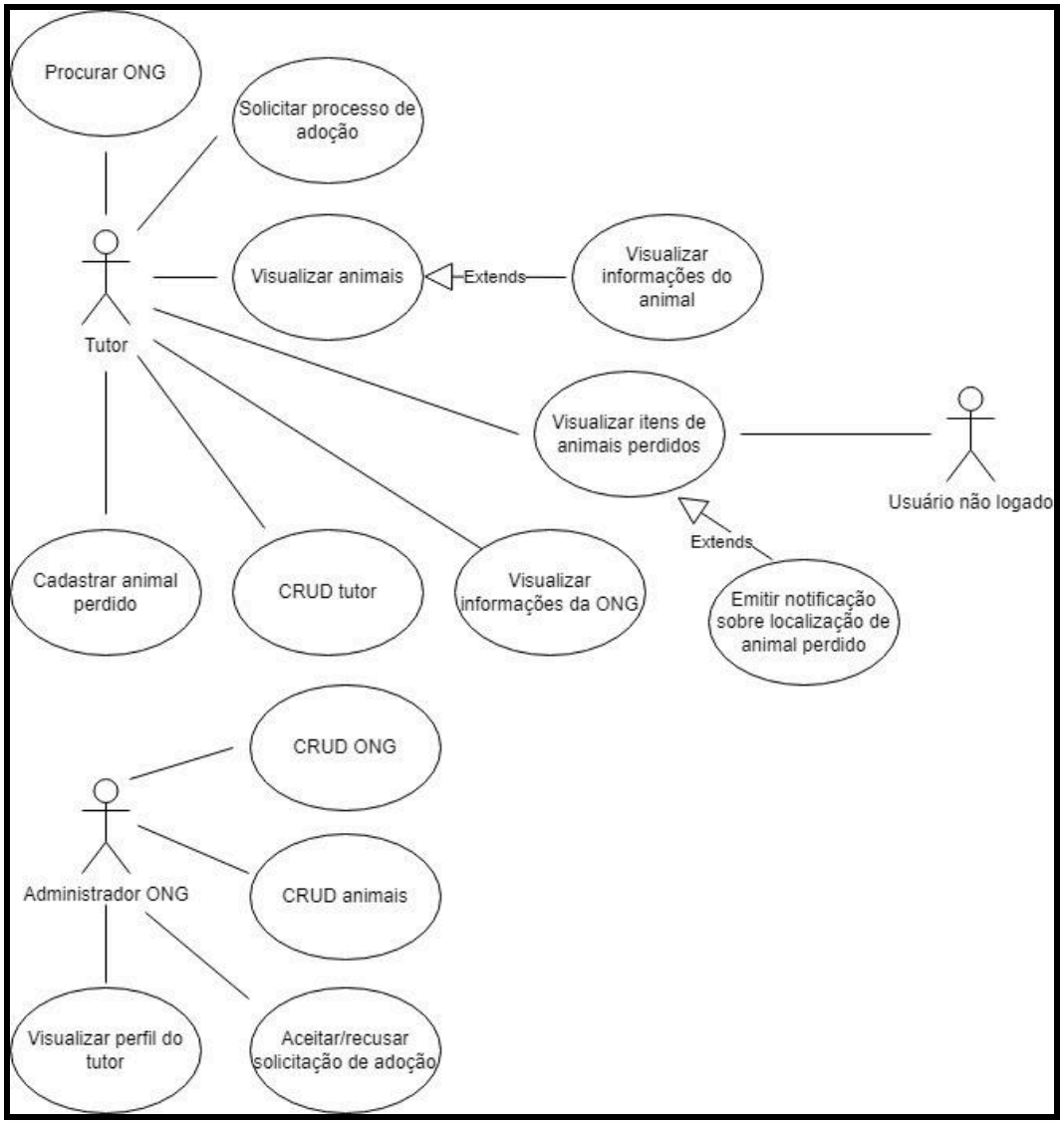

Figura 3. Diagrama de casos de uso. Autoria própria.

O diagrama de classe foi utilizado com o propósito de prover informações sobre os atributos, métodos e relacionamentos das classes presentes no sistema (Valente, 2023). A modelagem das classes pode ser observada na Figura 4, apresentando as classes do sistema, como a classe Tutor e a classe ONG, além dos relacionamentos dessas com as demais classes do sistema. Além disso, por meio deste diagrama é possível identificar como as funcionalidades presentes no diagrama de casos de uso estão sendo realizadas no sistema e quais classes podem ser afetadas por tal funcionalidade.

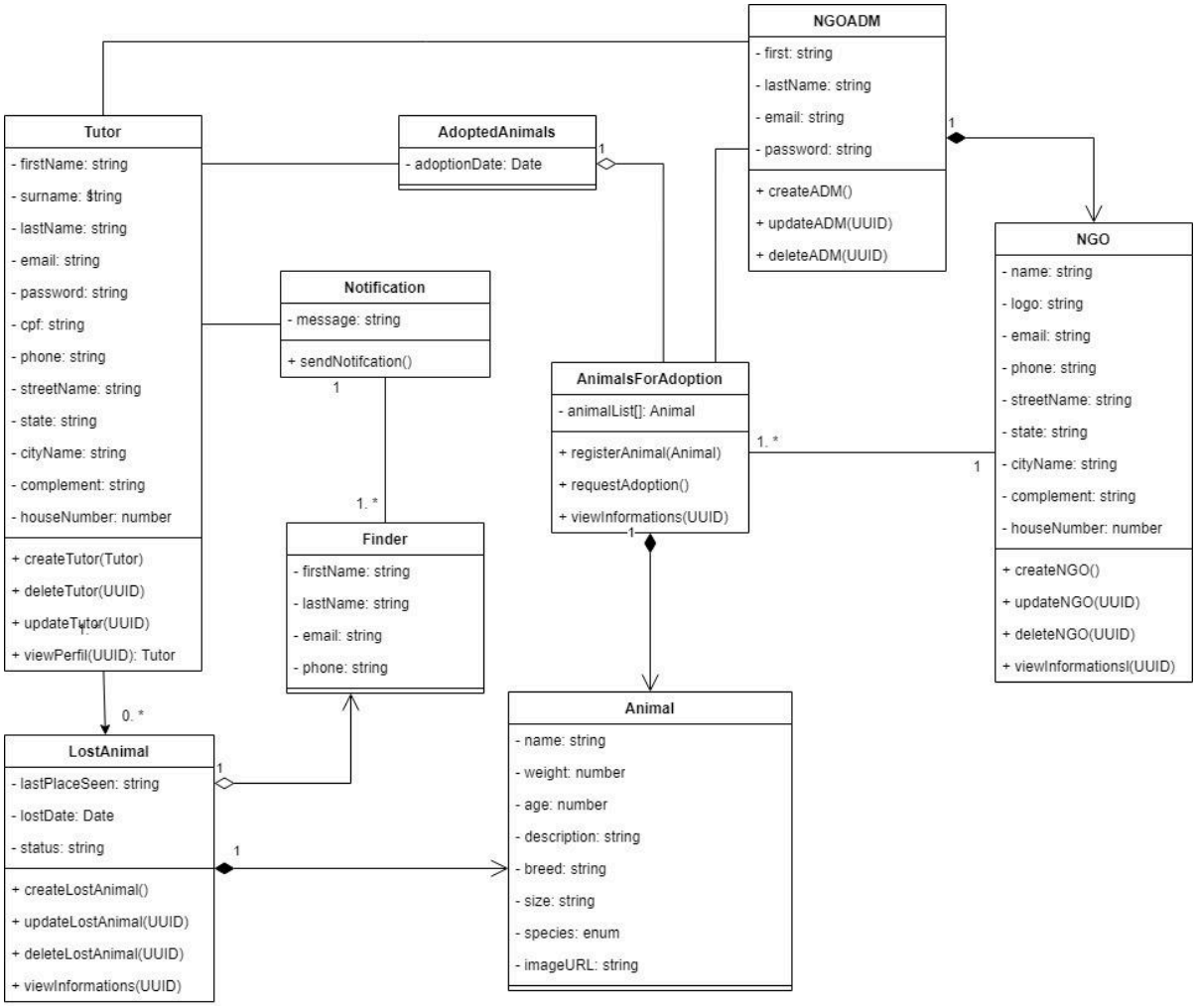

Figura 4. Diagrama de classes. Autoria própria.

5.3. Desenvolvimento *frontend*

De forma a auxiliar no desenvolvimento do código responsável pela criação da interface visual do sistema, protótipos das telas foram desenvolvidos utilizando a ferramenta *Figma*. Com isso, as telas da interface foram projetadas seguindo padrões definidos, como espaçamento entre componentes e arredondamento de botões, permitindo uma maior padronização em todos os elementos presentes nas interfaces da aplicação.

Com intuito de promover uma melhor experiência para o usuário, filtros foram adicionados para auxiliar o usuário na busca do animal que ele procura. Assim, o tutor pode procurar o animal que mais se adequa em suas preferências, escolhendo características como o sexo do animal, idade, gênero ou até mesmo uma ONG específica, facilitando o processo de adoção, visto que o tutor pode procurar uma organização específica que é localizada próxima a sua residência. Além disso, cada animal é apresentado em um *card* único, sendo que esse *card* apresenta uma foto do animal, seu nome, porte, sexo e local onde se encontra, além de suas

características, que são apresentadas por meio de *tags*. O protótipo para a interface pode ser visualizado na Figura 5.

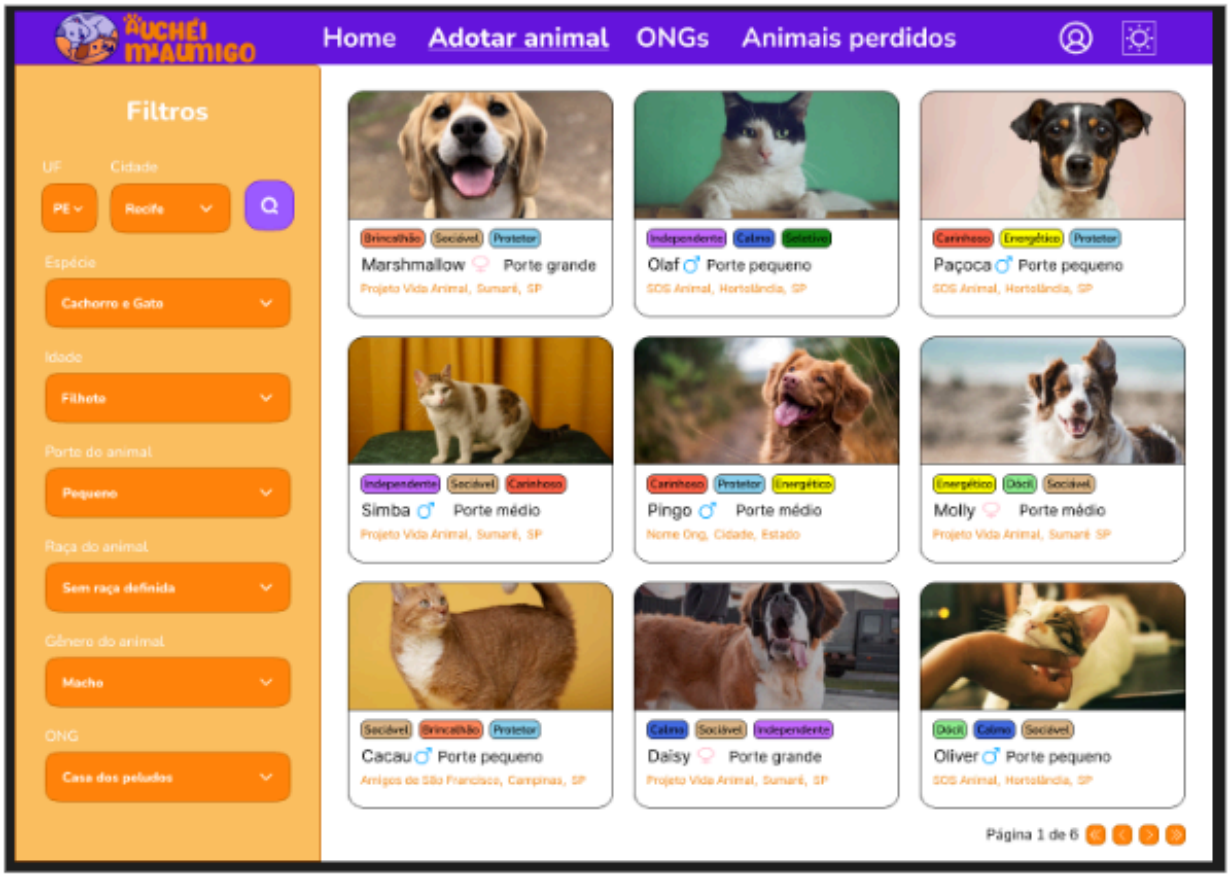

Figura 5. Página de busca de animais para adoção. Autoria própria.

Para implementar as interfaces inicialmente prototipadas no *Figma*, a linguagem *TypeScript* foi utilizada em conjunto com a biblioteca *React*. Essa combinação possibilita a modularização da interface, onde diferentes partes são organizadas como componentes independentes. Cada componente possui sua lógica, estilização e propriedades específicas, o que permite sua reutilização em diversas áreas da aplicação. Além disso, os componentes são projetados para gerenciar seus próprios estados, que podem ser alterados por interações do usuário, mudanças em outros componentes ou por suas próprias características, garantindo uma interface dinâmica.

Para a parte de estilização dos elementos da interface, foi utilizado o *framework TailwindCSS*. Esse *framework* possibilita que a estilização de cada elemento esteja presente no próprio componente, por meio de classes. Dessa forma, um arquivo CSS separado não se faz necessário para que a estrutura visual do componente seja alterada, mas sim que essa estrutura seja definida por meio de propriedades da própria classe.

No código que pode ser observado na Figura 6, há uma demonstração de como os componentes estão estruturados. Na linha 38, temos a definição de uma *div*, essa que representa uma contêiner genérico para o conteúdo. Na mesma linha, há a definição de uma *className*, sendo esse um atributo específico utilizado pelo *TailwindCSS* para definição da estilização de um conteúdo, seja ele um contêiner, imagem, texto ou qualquer outro elemento. Observando as definições de estilização apresentados na linha 38, temos as seguintes formatações: “mb-6 flex flex-col gap-4”. Dessa forma, o elemento, na interface terá, respectivamente: margem na parte inferior de 24px, display:flex, flex-direction:column e um gap de 16px. Tais medidas se dão pelo fato do *TailwindCSS* utilizar de unidades relativas e padronizadas em múltiplos de 4, assim todo os valores relativos utilizados são divididos por 4, então gap-4 seria na verdade gap-(4 * 4), nesse caso resultando em *gap* de 16px.

Por fim, na linha 43 é possível observar a presença do código “{tutor.firstName}”. Nesse caso, dependendo do valor recebido pelo objeto tutor, seu valor de *firstName* mudará e, conforme isso acontece, isso será refletido na interface. Para exemplificar o exemplo anterior, imaginemos que o usuário acesse um perfil de um tutor. Esse tutor está cadastrado no banco de dados com um identificador único. Dessa forma, quando há uma requisição para buscar os dados do tutor por meio de seu identificador, a resposta dessa requisição é armazenada como um objeto do tipo tutor. Assim, dependendo do atributo *firstName* que foi buscado no banco de dados por meio da requisição, um nome diferente será mostrado na página, permitindo com que a interface se adapte dependendo dos dados adquiridos.

```
38  <div className="mb-6 flex flex-col gap-4">
39    <div className="flex w-full gap-4">
40      <div className="flex flex-1 flex-col">
41        <label className="text-blue-900">Nome</label>
42        <p className="rounded border border-gray-300 px-3 py-2 text-blue-900">
43          {tutor.firstName}
44        </p>
45      </div>
46      <div className="flex flex-1 flex-col">
47        <label className="text-blue-900">Último nome</label>
48        <p className="rounded border border-gray-300 px-3 py-2 text-blue-900">
49          {tutor.lastName}
50        </p>
51      </div>
52    </div>
```

Figura 6. Código de um componente. Autoria própria.

A API do IBGE foi utilizada para obter os nomes dos Estados e cidades correspondentes. Para isso, é necessário configurar o *Next.js* para aceitar endereços externos. No arquivo .env, localizado na raiz da aplicação, está o endereço da API: "https://servicodados.ibge.gov.br/api/v1/localidades/estados". Com essa configuração, a biblioteca *axios* verifica e valida o acesso à API, garantindo sua disponibilidade na inicialização do sistema.

Ao selecionar uma unidade federativa, é possível escolher qualquer cidade desse Estado. A Figura 7 ilustra esse processo. Na linha 37, há a chamada para a API na rota "/", definida no arquivo .env e validada pelo *axios* . Após obter os Estados brasileiros, os dados são organizados alfabeticamente e armazenados na variável *states*. Na linha 46, se um Estadofor selecionado, ocorre uma nova chamada à API na rota "/municipios", que retorna as cidades do Estado escolhido.

```
31  export function LocationProvider({ children }: LocationProviderProps) {
32    const [states, setStates] = useState([])
33    const [selectedState, setSelectedState] = useState('')
34    const [cities, setCities] = useState([])
35
36    useEffect(() => {
37      ibgeAPI.get('/').then((response) => {
38        const sortedStates = response.data.sort((a: State, b: State) =>
39          a.sigla.localeCompare(b.sigla),
40        )
41        setStates(sortedStates)
42      })
43    }, [])
44
45    useEffect(() => {
46      if (selectedState) {
47        ibgeAPI.get(`/${selectedState}/municipios`).then((response) => {
48          const sortedCities = response.data.sort((a: City, b: City) =>
49            a.nome.localeCompare(b.nome),
50          )
51          setCities(sortedCities)
52        })
53      }
54    }, [selectedState])
```

Figura 7. Código do Location Context. Autoria própria.

5.4. Desenvolvimento de banco de dados

Durante essa etapa, a modelagem do banco de dados realizada no tópico 4.3 (Criação de diagramas) serviu como base para a implementação do banco que será utilizado no sistema. Para isso, o *Prisma* foi adotado para a definição dos modelos, que representam as tabelas responsáveis pela persistência dos dados, incluindo seus respectivos atributos. Esse processo é realizado por meio do arquivo schema.prisma. A Figura 8 ilustra a definição de um modelo e seus atributos, permitindo também a configuração de suas propriedades. No caso do modelo Tutor, ele possui o atributo id, que contém a propriedade @id, indicando que se trata do identificador da tabela. Além disso, a propriedade @default(uuid()) define que esse identificador será um UUID (Identificador Único Universal), gerado automaticamente a cada nova inserção na tabela.

Figura 8. Esquema do prisma. Autoria própria.

5.5. Desenvolvimento backend

Com intuito de fornecer as funcionalidades definidas na etapa de levantamento de requisitos, uma API REST foi desenvolvida para fornecer essas funcionalidades. Tais funcionalidades são disponibilizadas por meio de *endpoints*, esses recebendo parâmetros de acordo com o protocolo HTTP (*Hyper Text Transfer Protocol*) por meio de URIs (*Uniform Resource Identifier*- Identificador Uniforme de Recurso) ou mesmo no corpo de cada requisição.

Para a funcionalidade de criação de um tutor, por exemplo, temos um *controller* responsável pela rota de criação “POST /tutor”. Por meio da biblioteca *Zod* é possível realizar a validação do *body* da requisição recebida. Na Figura 9, temos a definição de um *body* válido na requisição, neste caso, cada valor recebido tem seu tipo definido e, em alguns casos, regras adicionais como o valor mínimo ou máximo de caracteres possível. Logo após, há o processo de parse, que permite validar e inferir o valor a uma variável, nesse caso *tutorInformations*.

```
21  const tutorRegisterBodySchema = z.object({
22    avatar: z.string(),
23    firstName: z.string().min(2),
24    lastName: z.string().min(2),
25    email: z.string().email(),
26    cpf: z.string().min(11).max(11),
27    password: z.string().min(8),
28    confirmPassword: z.string().min(8),
29    phone: z.string().min(10),
30    cep: z.string().min(8).max(8),
31    city: z.string(),
32    state: z.string().min(2).max(2),
33    houseNumber: z.string().max(4),
34    houseType: z.enum(['house', 'apartment']),
35    streetName: z.string(),
36  })
37
38  const tutorInfomations = tutorRegisterBodySchema.parse(request.body)
```

Figura 9. Validação do body da requisição. Autoria própria.

Após a requisição ser validada, o *controller* chama o caso de uso responsável pela criação de um tutor passando as informações recebidas. O caso de uso, por si, recebe essas informações, realiza as validações necessárias, como, por exemplo, a verificação se já há algum tutor com o email recebido, em caso positivo retorna erro. Após as validações, há a chamada do repositório responsável pela criação do tutor.

O código do *backend* foi projetado visando a separação das camadas da aplicação, cada uma com uma tarefa específica:

- *Controllers:* responsável por ser a interface de contato com o mundo externo, por meio de chamadas HTTP. Essa camada tem como função validar os dados recebidos, além de preparar os dados de resposta.
- Casos de uso: responsável por orquestrar os fluxos de trabalho da aplicação, gerenciando a lógica de como as operações devem ser realizadas, além de ser o local onde as regras de negócios definidas atuam. Essa camada atua por meio do recebimento de requisições dos *controllers*, aplicação das regras de negócio e interagindo diretamente com os repositórios para realização da tarefa solicitada. Seu principal benefício é o desacoplamento da lógica do negócio em uma camada distinta.
- Repositórios: responsável pela comunicação com a camada de persistência de dados, seja essa camada de persistência em memória ou em um banco de dados.

5.6. Testes

Nesta seção serão apresentados os testes realizados na aplicação.

5.6.1. Testes unitários

Testes unitários testam pequenas partes do sistema, como uma função ou uma classe. Esses são os tipos mais abundantes de testes realizados, devido ao fato de serem pouco acoplados a outras partes do sistema. Na Figura 10, o teste unitário em questão é utilizado para verificar se a senha do tutor no momento do registro está sendo criptografada.

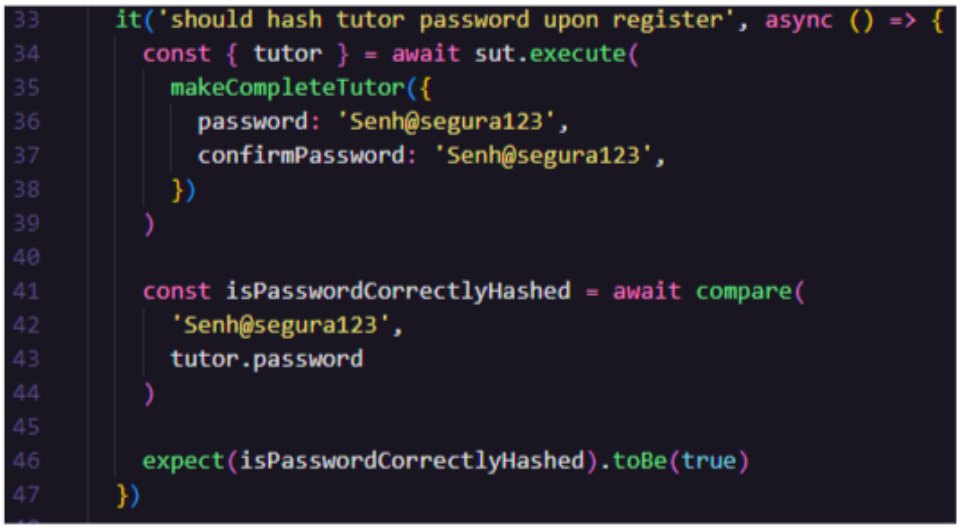

Figura 10. Teste unitário de hash da senha. Autoria própria.

Um importante ponto sobre testes unitários é o fato de que eles simulam um ambiente real. Com isso, testes unitários não devem interagir com o banco de dados, pois isso aumentaria consideravelmente o tempo de execução do mesmo. Dessa forma, repositórios em memória foram criados para simular a interação com o banco de dados, porém os dados são armazenados temporariamente na memória, em vez de serem persistidos em um banco real. Para isso, foi utilizado um vetor, inicialmente vazio. Novos objetos são criados seguindo um modelo pré-definido e, em seguida, adicionados a esse vetor. Essa abordagem permite testar funcionalidades sem a necessidade de um banco de dados real, facilitando o desenvolvimento e a execução de testes.

Ao todo, há 16 arquivos de testes e 58 testes unitários totais. Essa quantidade de testes leva, em média, 4 segundos para serem executados.

5.6.2. Testes de integração

Testes de integração tem como objetivo testar a interação entre duas ou mais partes do sistema. Dessa forma, é possível prever e confirmar que o *software* se comporta de maneira esperada após a execução de uma funcionalidade. No *frontend* do sistema, temos uma servidor *Mock*, ou seja,

um servidor com dados fictícios que simulam os dados recebidos pela aplicação após uma requisição. Com isso, é possível averiguar e validar os dados recebidos e apresentados pelo *frontend*, o que auxilia no desenvolvimento da interface da aplicação. Na Figura 11, é possível observar a aplicabilidade do servidor *Mock*, onde se é definido um tutor que será recebido após uma requisição. Dessa forma, é possível validar, por exemplo, que o primeiro nome do tutor apresentado na interface do usuário, nesse caso, seria Alexandrina.

Figura 11. Objeto tutor presente no servidor Mock. Autoria própria.

5.6.3. Testes *end-to-end*

Testes de *end-to-end* testam uma funcionalidade da aplicação por completo, desde o recebimento dos dados, o processamento, as regras de negócio aplicadas e a persistência desses dados. Na Figura 12, é possível observar um teste *end-to-end* para a criação de um tutor. No teste, um objeto tutor é feito a partir de uma *factory*, que seria uma classe própria para a criação de dados aleatórios e de estrutura padronizada. Após isso, é feita uma requisição para a rota de criação de um tutor, no corpo da requisição, há o objeto tutor previamente criado. A seguir, a verificação se o *status code* recebido como resposta da rota foi 201, confirmando a criação de um tutor. Por fim, há uma verificação se há um tutor com o email cadastrado no banco de dados, confirmando que todo o processo desde o recebimento até a persistência dos dados foi feito de maneira correta.

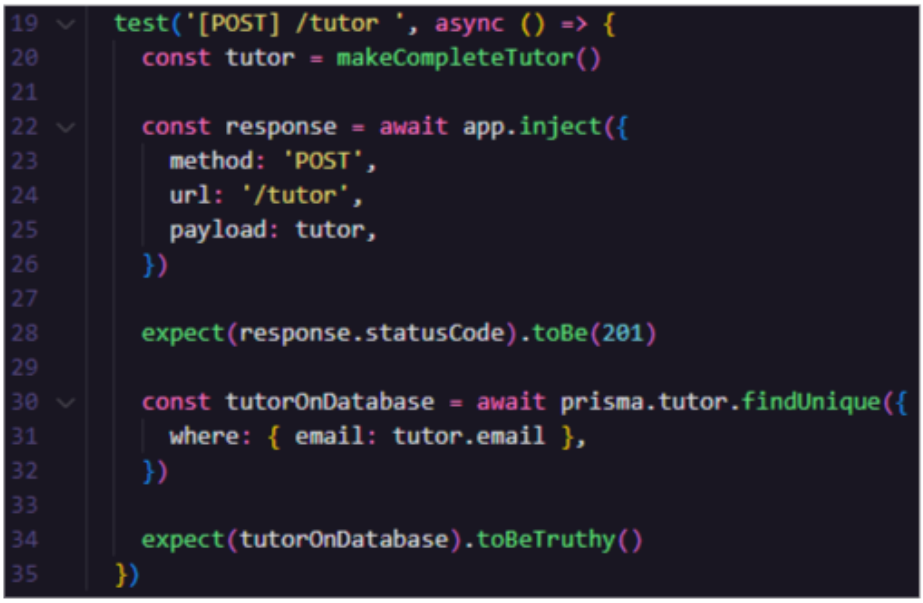

Figura 12. Teste *end-to-end* para criação de um tutor. Autoria própria.

Ao todo, há 21 arquivos de testes *end-to-end* e 23 testes. Essa quantidade de testes leva em média 50 segundos para serem realizados. Como podemos perceber, os testes *end-to-end* levam consideravelmente mais tempo para serem executados quando em relação aos testes unitários. Isso se dá ao fato de interagirem com todas as partes da aplicação, inclusive o banco de dados.

6. Conclusão

Este trabalho teve como objetivo desenvolver um *software* que auxiliasse no processo de adoção e localização de animais perdidos, de forma mais eficiente, rápida e prática, visando aumentar a quantidade de adoções e facilitar no processo de localização de um animal perdido. A proposta do sistema veio a partir da facilidade obtida por meio de sistemas *webs* para a realização de atividades. Com o desenvolvimento do Auchei Miaumigo, esse processo de adoção de animais pode ser feito de maneira mais rápida e fácil, proporcionando ao usuário uma lista de animais disponíveis, assim como ONGs localizadas próximas a ele. Além disso, uma lista de animais desaparecidos influencia na facilidade da localização de um animal, fazendo com que mais tutores possam reencontrar seus amigos desaparecidos.

O presente trabalho possibilitou a implementação de diversos *frameworks*, como, por exemplo, o *fastify*, utilizado no *backend*, e o *Next.js*, utilizado no *frontend*. Além disso, diversos padrões de projetos puderam ser utilizados, sendo eles o *Factory Pattern*, utilizado no *backend* para criação dos casos de uso e também para a criação dos objetos utilizados nos testes; o *Repository Pattern*, que permite o encapsulamento da parte lógica do acesso a camada de persistência de dados, além de permitir o uso de injeção de dependência na camada de negócio, ou seja, nos casos de uso.

Como trabalho de conclusão de curso, este trabalho permitiu o emprego de conhecimentos adquiridos nas disciplinas do Curso Superior em Análise e Desenvolvimento de Sistemas, tais como: Algoritmos e Programação; Linguagem de Programação; Banco de Dados; Engenharia de *Software*; Análise Orientada a Objetos; Arquitetura de *Software*; Programação Orientada a Objetos; Desenvolvimento *Web*; Projeto de Sistemas e Qualidade de *Software*.

Como trabalhos futuros podemos citar a adição de funcionalidades como a geolocalização de ONGs e animais e também a possibilidade da integração do sistema com os sistemas já existentes em petshops e ONGs.

Referências

Amazon, W. S. (2023). "o que é api?". Acesso em: 18 set. 2024.

Caetano, E. C. S. (2010). As contribuições de taa – terapia assistida por animais à psicologia. Acesso em: 13 ago. 2024.

Cejas, Fernando. "Architecting Android: The Clean Way." Fernando Cejas, 3 setembro 2014,

Cohn, M. (2011). Desenvolvimento de software com Scrum: aplicando métodos ágeis com sucesso. Bookman, Porto Alegre.

Fastify (2024). Fastify: Framework web rápido e de baixo overhead para node.js. Acesso em: 18 set. 2024.

Félix, R. o. (2016). Teste de software. Pearson, São Paulo. E-book. Disponível em: https://plataforma.bvirtual.com.br. Acesso em: 30 jan. 2025.

Hunt, A. and Thomas, D. (2003). Pragmatic Unit Testing: In Java with JUnit. The Pragmatic Bookshelf, Estados Unidos.

IBGE (2024). Api de localidades. Acesso em: 18 set. 2024.

LOCPET (2022). Locpet: aplicativo para localização de animais. Acesso em: 12 ago. 2024.

MDN, W. D. (2024). Json: Notação de objetos javascript. Publicado em: 29 de julho de 2024, Acesso em: 18 set. 2024.

Microsoft (2024). Descrição da normalização do banco de dados. Acesso em: 09 ago. 2024.

Next.js (2024). Next.js: Framework para react que facilita a criação de aplicações web. Acesso em: 18 set. 2024.

Node.js (2024). Node.js: Javascript runtime. Acesso em: 17 set. 2024.

Oracle (2020). "what is a database?". Acesso em: 18 set. 2024.

Pastori, Érica Onzi e Matos, L. G. (2015). Da paixão à “ajuda animalitária”: o paradoxo

do “amor incondicional” no cuidado e no abandono de animais de estimação. Caderno Eletrônico de Ciências Sociais, 3(1):112–132. Acesso em: 12 jul. 2024.

Pacheco, M.Desenvolvimento WEB: HTML, CSS E JAVASCRIPT para iniciantes.[S.l.: s.n.], 2023. E-book Kindle.

Patronek, G. J., Glickman, L. T., Beck, A. M., McCabe, G. P., and Ecker, C. (1997). Risk factors for relinquishment of cats to an animal shelter. Journal of the American Veterinary Medical Association, 209(3):582–588.

Petz, A. (2024). Adote petz: Adoção de animais. Acesso em: 12 ago. 2024.

Prisma (2025). Prisma – next-generation node.js and typescript orm. Acesso em: 29 jan. 2025.

Puente, B. (2022). Brasil tem quase 185 mil animais resgatados por ongs, diz instituto. Acesso em: 12 jul. 2024.

React (2024). React: Biblioteca javascript para criação de interfaces de usuário. Acesso em: 18 set. 2024.

Severino, R. O. (2023). Importância das organizações não governamentais (ongs) de animais e seus impactos no meio ambiente e urbano. Acesso em: 12 jul. 2024.

Sommerville, I. (2016). Engenharia de Software. Pearson, 10 edition. Acesso em: 25 nov. 2024.

Tavares, M. V. L. (2024). Petadote: Aplicação web que auxilie na adoção e divulgação de animais abandonados. Acesso em: 12 ago. 2024.

TypeScript (2024). Typescript: Javascript with syntax for types. Acesso em: 17 set. 2024.

UNESCO (1978). Declaração universal dos direitos dos animais. Proclamada em sessão realizada em Bruxelas, Bélgica. Acesso em: 15 jul. 2024.

V8 (2024). V8: Engine de javascript de código aberto. Acesso em: 17 set. 2024.

Valente, M. T. (2023). Engenharia de software moderna: princípios e práticas para desen-desenvolvimento de software com produtividade. Acesso em: 26 jul. 2024. Disponível em: https://engsoftmoderna.info/.

Apêndice I

Visando um foco no domínio da aplicação, a arquitetura da aplicação foi baseada na *use-case architecture* (arquitetura de casos de uso) devido ao seu foco nas regras de negócio e entidades presentes no sistema. Assim como Fernando Cejas (2014), adaptou a *Use-Cases Architecture* para o desenvolvimento *Android*, este projeto adaptou as camadas dessa arquitetura visando suprir as necessidades da aplicação. A arquitetura de casos de uso tem foco em camadas no sistema, tendo 4 principais camadas: UI, *Presenters*, *Use Cases*, *Entities.* Na Figura 13, tais camadas podem ser observadas.

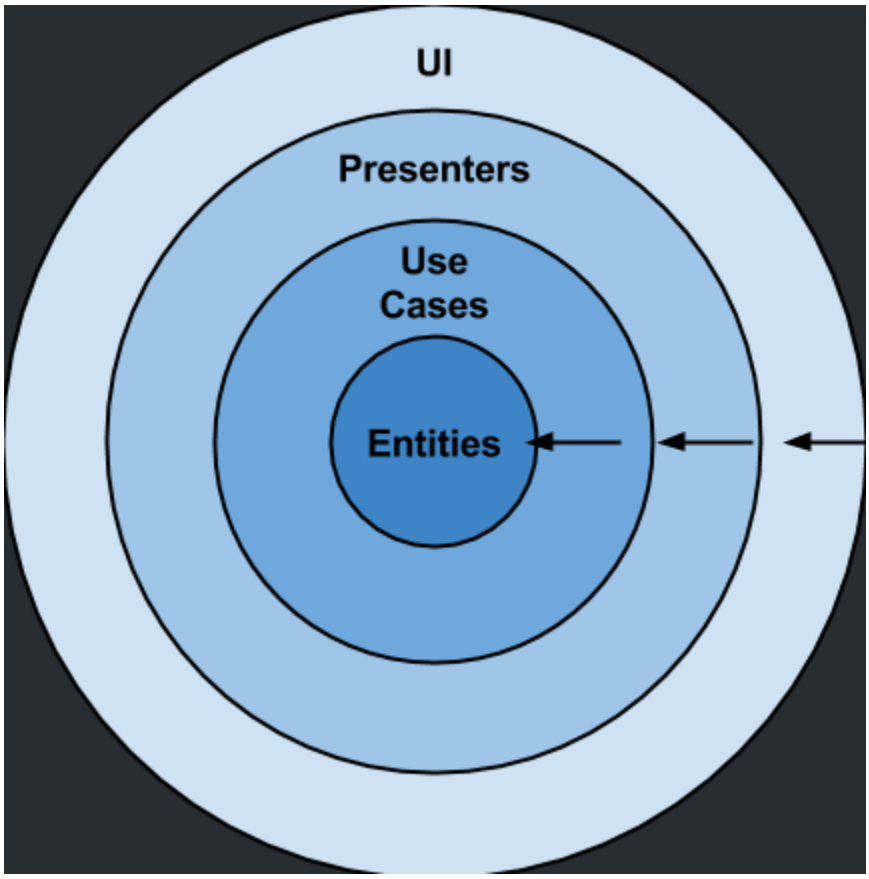

Figura 13. Arquitetura de casos de uso. Disponível em: https://fernandocejas.com/2014/09/03/architecting-android-the-clean-way/ (Adaptado)

Utilizando a arquitetura e fazendo as adaptações conforme o necessário, temos as seguintes camadas:

Camada UI

A camada UI (*User Interface*) é responsável por capturar as entradas do usuário, exibir as saídas e garantir uma experiência de uso eficiente. Ela recebe as interações do usuário e as repassa para

a camada de *Presenters*, que valida e formata os dados antes de enviá-los para os Casos de Uso. Após o processamento, a resposta retorna para a UI de forma adaptada para exibição. Nesta camada da arquitetura, todo o *frontend* da arquitetura está localizado.

Camada *Presenters (Controllers)*

A camada *Presenters* tem como função preparar os dados que serão exibidos na camada UI e garantir que as respostas dos casos de uso sejam apresentadas de forma adequada. Ela recebe as informações da camada de Casos de Uso, adapta os dados para o formato esperado pela interface e os retorna para exibição. Além disso, pode realizar validações e formatações antes de encaminhar as solicitações para os casos de uso. Essa separação mantém a lógica de apresentação independente da interface, facilitando mudanças e reutilização.

Essas validações e formatações são realizadas por meios de *Controllers*, que tem como função "conversar" diretamente com os casos de uso responsável pela operação solicitada. Essa solicitação é feita por meio de requisições *HTTP realizadas pela camada UI.*

Camada *Use cases*

A camada *Use Cases* (Casos de Uso) é responsável por coordenar a lógica de negócio da aplicação. Essa camada recebe solicitações da camada de *Presenters* (*Controllers*), executa as regras de negócio necessárias e interage com a camada de *Entities* para processamento de dados. Após a execução, retorna o resultado para os *Presenters* (*Controllers*), que preparam a resposta para a UI. Essa camada garante que a lógica de aplicação fique centralizada e independente de detalhes de implementação, facilitando testes e manutenção.

De forma a proteger a integridade do sistema, essa é a única camada que pode solicitar diretamente acesso aos dados armazenados no banco de dados. Outro aspecto dessa camada é a independência de bibliotecas e *frameworks,* o funcionamento desse nível de forma independente dos demais.

Camada *Entities (Repositories)*

A camada *Repositories* tem como função fornecer acesso e persistência dos dados. Ela interage com bancos de dados para salvar ou recuperar informações necessárias para a execução dos Casos de Uso. Os *Repositories* abstraem a lógica de acesso aos dados, permitindo que a camada de Use Cases se concentre apenas nas regras de negócio sem se preocupar com detalhes de implementação de armazenamento. Essa camada garante que o sistema seja flexível, permitindo trocas de tecnologia de persistência sem afetar a lógica de aplicação.

Os repositórios dessa camada atuam como um contrato que deve ser seguidos para que um repositório construído, siga os padrões previamente definidos. No sistema há dois tipos de repositórios diferentes, sendo eles os repositórios em memória, que guardam os dados inseridos na memória do computador por meio de um vetor, permitindo que os testes unitários sejam

realizados de forma rápida e eficiente; e os repositórios do *Prisma,* que ativamente interagem com o banco de dados para realização da persistência e busca de dados.

Documento Digitalizado Público

Artigo Final de TCC do Yan Valadares

Assunto: Artigo Final de TCC do Yan Valadares
Assinado por: Leandro Ledel
Tipo do Documento: Estudo Técnico
Situação: Finalizado
Nível de Acesso: Público
Tipo do Conferência: Cópia Simples

Documento assinado eletronicamente por:

- **Leandro Camara Ledel**, **PROFESSOR ENS BASICO TECN TECNOLOGICO**, em 10/03/2025 07:46:27.

Este documento foi armazenado no SUAP em 10/03/2025. Para comprovar sua integridade, faça a leitura do QRCode ao lado ou acesse https://suap.ifsp.edu.br/verificar-documento-externo/ e forneça os dados abaixo:

Código Verificador: 1959542
Código de Autenticação: 2af7c7093a