UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
CURSO SUPERIOR DE TECNOLOGIA EM ANÁLISE E DESENVOLVIMENTO DE SISTEMAS

VOMISLEI MATEUS BALENA

SISTEMA WEB PARA GESTÃO IMOBILIÁRIA

TRABALHO DE CONCLUSÃO DE CURSO

PATO BRANCO
2019

VOMISLEI MATEUS BALENA

SISTEMA WEB PARA GESTÃO IMOBILIÁRIA

Trabalho de Conclusão de Curso de graduação, apresentado à disciplina de Trabalho de Conclusão de Curso, do Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas, da Universidade Tecnológica Federal do Paraná, Campus Pato Branco, como requisito parcial para obtenção do título de Tecnólogo.

Orientador: Prof. Vinicius Pegorini

PATO BRANCO
2019

Ministério da Educação
Universidade Tecnológica Federal do Paraná
Campus Pato Branco
Departamento Acadêmico de Informática
Curso de Tecnologia em Análise e Desenvolvimento de Sistemas

TERMO DE APROVAÇÃO

TRABALHO DE CONCLUSÃO DE CURSO

Sistema Web para Gestão Imobiliária

POR

VOMISLEI MATEUS BALENA

Este trabalho de conclusão de curso foi apresentado no dia 02 de dezembro de 2019, como requisito parcial para obtenção do título de Tecnólogo em Análise e Desenvolvimento de Sistemas, pela Universidade Tecnológica Federal do Paraná. O acadêmico foi arguido pela Banca Examinadora composta pelos professores abaixo assinados. Após deliberação, a Banca Examinadora considerou o trabalho aprovado.

Banca examinadora:

Prof. MSc
Vinicius Pegorini

Prof. Dr.
Ives Renê Venturini Pola

Profª Drª
Viviane Dal Molin de Souza

Prof. Dr. Edilson Pontarolo
Coordenador do Curso de Tecnologia em Análise e Desenvolvimento de Sistemas

Profª Drª Mariza Miola Dosciatti
Responsável pela Atividade de Trabalho de Conclusão de Curso

A Folha de Aprovação assinada encontra-se na Coordenação do Curso.

RESUMO

Empresas dos mais diversos segmentos têm feito uso crescente de ferramentas que visam auxiliar o processo de gestão e prospecção de novos clientes. No ramo imobiliário não é diferente, no cenário atual de ascensão de negociações de imóveis, os sistemas que utilizam plataforma web têm se tornado muito eficazes para suprir essas necessidades. Com o objetivo de facilitar a gestão financeira, contábil e pessoal de empresas do ramo imobiliário, neste trabalho foi desenvolvido um sistema web para gestão imobiliária, que visa proporcionar rapidez no acesso às informações da empresa. O sistema possibilita aos usuários cadastrar novos clientes e seus imóveis, permite criar e gerenciar negociações com seus clientes, além disso, possibilita um controle financeiro, e acesso a imóveis disponíveis através de um site.

Palavras-chave: Sistema web. Angular. Spring. Sistema de gestão para imobiliárias.

ABSTRACT

Companies of the most diverse segments have made an increasing use of tools aimed at assisting the process of management and prospection of new clients. In the current scenario of real estate negotiations, a web system platform had become effective to meet these needs, easing the financial, accounting and personnel management of real estate companies. In this work a web system for real estate management was developed, aiming at providing quick access to the company information. The system also allows users to register new customers and their properties, create and manage negotiations with their customers, and besides that, the possibility of financial control and access to available properties through a website.

Keywords: Web system. Angular. Spring. Management system for real estate.

LISTA DE QUADROS

LISTA DE FIGURAS

LISTAGEM DE CÓDIGOS

LISTA DE SIGLAS E ACRÔNIMOS

AJAX	Asynchronous Javascript and XML
HTML	HyperText Markup Language
IDE	Integrated Development Environment
RIA	Rich Internet Application
REST	Representational State Transfer
SECOVI	Sindicato das Empresas de Compra, Venda, Locação, Administração de Imóveis e dos Condomínios Residenciais e Comerciais
SPA	Simple-Page Application
URL	Uniform Resource Locator

SUMÁRIO

1 INTRODUÇÃO

Neste capítulo são apresentadas as considerações iniciais do trabalho, os seus objetivos e a justificativa, finalizando com a estrutura do documento.

1.1 CONSIDERAÇÕES INICIAIS

Cada vez mais as empresas buscam utilizar tecnologia para potencializar os processos diários em termos de velocidade e precisão. Em um âmbito empresarial, essas tecnologias servem principalmente para garantir o controle de pessoas, financeiro e contábil da companhia, assim agregando informações sobre recebimento de receitas, pagamento de despesas.

Segundo Bazotti e Garcia (2006, p. 4), o sucesso das empresas atualmente está totalmente vinculado a velocidade em que as informações são assimiladas e pela rapidez em que são tomadas as decisões. Os componentes que fundamentam a tecnologia de informação são os grandes precursores desse sucesso.

Diante da busca por soluções mais adequadas para gerenciar seus produtos ou serviços, os sistemas web tem sido uma das soluções mais procuradas por pequenas, médias e até grandes empresas, por serem ferramentas modernas que aliam praticidade com desempenho. Os sistemas web permitem integrar todos os dados da empresa, aproximando setores, diminuindo custos e minimizando os erros.

Segundo a SECOVI (2018, p. 4), o resultado de pesquisa de mercado do primeiro semestre de 2018 na cidade de São Paulo – SP, o mercado de venda de imóveis apresentou um aumento de 52,1% em relação ao mesmo período do ano de 2017. Diante dessa ascensão de negociações no ramo imobiliário, a demanda por soluções para suprir as necessidades de melhorias na gestão das empresas tende a aumentar.

Nesse contexto, uma aplicação web onde o acesso ao sistema ocorre via Internet permite facilitar aos colaboradores da empresa manter em dia informações de clientes, imóveis, contratos, contas, entre outras coisas, facilitando assim os processos de gestão da empresa. A disponibilidade de um site, irá possibilitar a criação de um vínculo maior entre a empresa e seus clientes, proporcionando um fortalecimento do nome da companhia. A aplicação web será desenvolvida utilizando o conceito de “Aplicação de Internet Rica”, concentrando a maior parte do processamento da interface no navegador e mantendo os dados no servidor de aplicação.

1.2 OBJETIVOS

A seguir serão apresentados o objetivo geral e específicos do presente trabalho.

1.2.1 Objetivo Geral

Desenvolver um sistema web para possibilitar a gestão de empresas que atuam no ramo imobiliário.

1.2.2 Objetivos Específicos

Dentre os objetivos específicos do trabalho, destacam-se:

- Possibilitar o acompanhamento das receitas da empresa.
- Permitir a unificação das informações dos imóveis, clientes e receitas em uma só solução.
- Possibilitar a disponibilização e acesso a informações de imóveis disponíveis.
- Facilitar a geração de documentação para fins de imposto de renda.
- Facilitar a prospecção de novos clientes através de um site integrado ao sistema.
- Oferecer um sistema de propostas de preços nas vendas e locações.

1.3 JUSTIFICATIVA

Este projeto tem como objetivo fornecer um software de gestão imobiliário agregando qualidade e agilidade nos processos administrativos. Percebendo a necessidade em muitas empresas de pequeno e médio porte em possuir um software de gestão que facilite a união das informações ora espalhadas, em um só lugar, assim gerando economia de tempo e reduzindo retrabalho.

Diante dessas necessidades, surge a oportunidade de desenvolver um sistema para plataforma web. Um sistema web não precisa ser instalado em cada máquina que será acessado, seja o acesso via computador, celular ou tablet, dessa forma reduzindo investimentos na implantação do sistema.

Além desses fatores contribuindo com a prospecção de novos clientes possibilitado pelo site onde pode-se ter maior facilidade de divulgação, aliado a um portal para clientes onde se teria um maior contato com os clientes.

O principal fator que levou a escolha desse tema foi devido a outros ramos de negócio utilizarem um modelo de negócio muito parecido, por exemplo, locadoras e revendas de automóveis, onde essa aplicação poderia ser facilmente ajustada para suprir essas necessidades.

1.4 ESTRUTURA DO TRABALHO

O Capítulo 2 apresenta uma contextualização teórica sobre aplicações web, e informações sobre o ramo imobiliário.

No Capítulo 3 são apresentados os materiais e método que serão utilizados para a realização deste trabalho.

No Capítulo 4 são apresentados os resultados do desenvolvimento deste trabalho.

No Capítulo 5 está a conclusão, seguida das referências utilizadas no texto.

2 REFERENCIAL TEÓRICO

Este capítulo apresenta o referencial teórico trazendo embasamento sobre conceitos relacionados às aplicações Internet denominadas *Rich Internet Application*. Esse tipo de aplicação caracteriza o conjunto de ferramentas de desenvolvimento e funcionalidades que serão utilizadas no desenvolvimento deste trabalho.

2.1 APLICAÇÕES INTERNET RICAS

Diferentemente das aplicações web baseadas em tags, tal como a *HyperText Markup Language* (HTML), as *Rich Internet Applications* (RIA) também conhecidas como Aplicações de Internet Ricas integram novas tecnologias, entre essas novas tecnologias estão *Asynchronous Javascript and XML* (AJAX), que proporciona um algo grau de interatividade suportando amplamente o novo tipo de aplicações para Internet (AGUSTIN, 2015).

As RIAs são geralmente associadas à aplicações desenvolvidas para plataforma web que fornecem os recursos e funcionalidades dos aplicativos tradicionais desenvolvidos para plataforma *desktop* (MELIÁ, 2010). O termo RIA foi primeiramente apresentado em um *white paper* da Micromedia no ano de 2002 (JEREMY, 2002).

Em termos de interação, as RIAs primam por uma interface eficaz com o usuário, com melhor responsividade e mais recursos e funcionalidades de interação visando prover uma interface rica com o usuário. Elas tipicamente dependem fortemente de tecnologias no lado cliente, combinadas com comunicação assíncrona para atender a esses requisitos (CASTELEYN; GARRIGÓS; MAZÓN, 2014).

As RIAs, além de apresentarem uma melhor usabilidade, permitem melhorar o desempenho da aplicação. Isso é possível ao reduzir significativamente o número de requisições feitas para o servidor, priorizando o processamento no lado cliente, reduzindo assim a atualização da página e o tráfego na Internet (BERNARDI; DI LUCCA; DISTANTE, 2014).

Com o aumento do número de informações geradas pelas operações das empresas, se faz necessário um sistema mais robusto para analisar e organizar os dados do negócio para acompanhar sem solavancos o crescimento pelo qual ele passa. Neste caso, a computação em nuvem aliada às RIAs, auxiliam as empresas a interpretar e tomar decisões estratégicas relacionadas ao mercado.

Classificação simplificada das características de RIA do lado cliente (MELIÁ *et al*., 2010):

- Trabalho *offline* - possibilidade de trabalhar enquanto desconectado, por meio de download da lógica de negócios e os dados no lado do cliente. Esse recurso requer a implementação de uma lógica de negócios e um mecanismo de armazenamento.
- Armazenamento - as RIAs fornecem novas facilidades de armazenamento no lado do cliente, manipulando os dados que vêm do servidor, podendo ser permanente ou volátil.
- *Cache* – o cliente RIA tem a capacidade de manter as informações do servidor durante um período de tempo, melhorando o desempenho da aplicação e a responsividade da interface do usuário. O armazenamento em cache possui um atributo de busca que suporta valores imediatos ou lentos.
- Lógica de negócios – o cliente RIA possui uma capacidade de processo aprimorada, assim permitindo a realização de processos complexos. Essa característica possui um atributo *location* que determina se a lógica de negócio é do cliente, do servidor ou uma mistura de ambos.
- Manipulação de eventos - As RIAs têm uma lógica baseada em eventos entre diferentes componentes de interface do usuário que podem ser sincronizados com o barramento de eventos centralizado ou com o padrão de observador descentralizado.
- Validação – As RIAs podem conter regras de validação para entradas de usuários, bem como para regras de negócios no cliente.
- Template – Possibilita suportar a criação de visualizações e o layout da apresentação em tempo de execução.
- Plataforma - Determina qual a plataforma utilizada para implementar a camada do cliente.

As aplicações RIAs focam principalmente na melhora do lado cliente da aplicação, ou seja, a aplicação que irá rodar no navegador no usuário. Para o desenvolvimento de uma aplicação web o lado servidor da aplicação também é importante, pois é a parte da aplicação que será responsável por manipular os dados armazenados no banco de dados. Uma das maneiras de escrever uma aplicação web que roda no lado servidor é utilizando a arquitetura *Representational State Transfer* (REST).

2.2 REST

O REST é um estilo de desenvolvimento de web services que foi apresentado pela primeira vez na tese de doutorado de Roy Fielding. Este, por sua vez, é co-autor do HTTP, que é um dos protocolos mais utilizados no mundo, o HTTP. Assim, é notável que o protocolo REST é guiado pelo que seriam as boas práticas de uso de HTTP:

- Uso adequado dos métodos HTTP;
- Uso adequado de URL's;
- Uso de códigos de status padronizados para representação de sucessos ou falhas;
- Uso adequado de cabeçalhos HTTP;
- Interligações entre vários recursos diferentes(SAUDATE, 2013).

Englobando as restrições fundamentais sobre componentes, conexões e dados que definem a base da arquitetura web, o REST é uma abstração dos elementos arquitetônicos dentro de um sistema hipermídia distribuído. O REST ignora os detalhes da implementação do componente e da sintaxe do protocolo priorizando as funções dos componentes, restrições sobre sua interação com outros componentes e interpretação de elementos de dados significativos (FIELDING, 2000).

A arquitetura de hipermídia distribuída possui três maneiras de tratamento de dados, que são elas:

- Renderizar os dados onde estão localizados e enviar uma imagem de formato fixo para o destinatário.
- Encapsular os dados com um mecanismo de renderização e enviar ambos para o destinatário.
- Enviar os dados brutos para o destinatário juntamente com os metadados que descrevem o tipo de dados, para que o destinatário possa escolher seu próprio mecanismo de renderização.

O REST fornece um híbrido de todas as três opções, focando em um entendimento compartilhado dos tipos de dados com metadados, mas limitando o escopo do que é revelado em uma interface padronizada. Os componentes REST comunicam-se transferindo uma representação de um recurso em um formato que corresponda a um conjunto em evolução de tipos de dados padrão, selecionando dinamicamente com base nos recursos ou necessidades do destinatário e na natureza do recurso. Se a representação está no mesmo formato da origem bruta ou é derivada da origem, ela permanece oculta por trás da interface. Os benefícios do

estilo de objeto móvel são aproximados, enviando uma representação que consiste em instruções no formato de dados padrão de um mecanismo de renderização encapsulado (FIELDING, 2000).

3 MATERIAIS E MÉTODO

Este capítulo apresenta os materiais e o método utilizado para a realização deste trabalho. Os materiais estão relacionados às tecnologias e ferramentas utilizadas e o método apresenta a sequência das principais atividades realizadas.

3.1 MATERIAIS

O Quadro 1 apresenta as ferramentas e as tecnologias que foram utilizadas para modelar e implementar o sistema.

Quadro 1 – Ferramentas e tecnologias utilizadas

Ferramenta / Tecnologia	Versão	Finalidade
Angular	5.0.0	Framework para desenvolvimento do lado cliente da aplicação
Apache Tomcat		Servidor de aplicações Java para web que implementa as tecnologias JavaServlets e JavaServer Pages.
Eclipse	Oxygen.2	Ambiente de desenvolvimento para o lado servidor da aplicação.
Java	10	Linguagem de programação.
PgAdmin4	2.0	Ferramenta de interface gráfica para o controle e desenvolvimento de bancos de dados PostgreSQL.
PostgreSQL	10.5	PostgreSQL é um sistema de banco de dados *open source* e objeto-relacional.
PrimeNG	8.1.0	Biblioteca de componentes.
Spring Framework	2.0.0	Framework para desenvolvimento do lado servidor da aplicação.
Spring Boot	1.4.0.	O Spring boot visa facilitar a criação de aplicações autônomas, baseadas em Spring.
Spring Security	4.1.1	Spring security é uma estrutura que fornece autenticação e autorização para aplicativos baseados em Java.
Spring Data JPA	1.4.0	A Spring Data JPA tem como proposta melhorar e facilitar a implementação de camadas de acesso a dados.
Visual Paradigm Community Edition	13.1	Modelagem de diagramas, e mapeamento de objetos relacionais.
Visual Studio Code	1.21.1	Ambiente de desenvolvimento para o lado cliente da aplicação.

Fonte: Autoria própria.

3.2 MÉTODO

As principais atividades realizadas para o desenvolvimento deste trabalho foram:

- Levantamento de requisitos – O levantamento de requisitos foi realizado a partir de visita a *sites* de locadoras e com base no entendimento das regras de negócio a partir de entrevista com funcionários do ramo, identificando as necessidades dos usuários que utilizam esse tipo de aplicação.
- Análise e Projeto – Com base nos requisitos definidos, foram elaborados os diagramas de casos de uso do sistema. Ao gerar os casos de uso foi possível obter as informações essenciais para que fossem gerados o diagrama de entidade e relacionamento, diagrama de classes e a modelagem do banco de dados.
- Desenvolvimento – Na etapa de desenvolvimento, a implementação foi realizada utilizando o *Integrated Development Environment* (IDE) Eclipse para o desenvolvimento do lado servidor e *Visual Studio Code* para o desenvolvimento do lado cliente do sistema.
- Testes – Para validar as funcionalidades do sistema, os testes foram realizados sem um plano de testes específico, onde foram realizados os visando a identificação de erros de codificação, identificação de melhorias em funcionalidades do sistema e a verificação do atendimento dos requisitos definidos e modelados.

4 RESULTADOS

Este capítulo apresenta o resultado deste trabalho que é o desenvolvimento de um sistema *web* para gestão imobiliária. Nele é apresentado o escopo e modelagem do sistema, a apresentação e listagens de códigos implementados.

4.1 ESCOPO DO SISTEMA

O sistema desenvolvido gerencia os processos administrativos e financeiros de uma empresa imobiliária, dentre eles o recebimento de aluguéis, compra e venda de imóveis e negociação com clientes. O sistema permite gerenciar o fluxo de caixa, contas a pagar e a receber, é possível realizar o controle de recebimentos e repasses de locações anuais da imobiliária.

Os usuários do sistema acessam determinadas funcionalidades do sistema de acordo com seu nível de permissão. O acesso ao sistema é realizado por meio da Internet, ou seja, via navegador web, sendo possível ser acessado por qualquer dispositivo que possua essa tecnologia. Após estar autenticado no sistema, será exibida uma página com informações para fácil e rápida visualização, tais como contratos de locações realizados e encerrados por mês nos últimos meses, contas a pagar e a receber no mês, número de intermediações de vendas por mês nos últimos meses.

No cadastro de clientes, detalham-se as informações dos clientes baseado em seus interesses de negócios. No cadastro dos imóveis está incluído informações sobre o imóvel, este vinculado a um cliente, incluindo a localização, onde posteriormente será utilizada nas informações do imóvel no site.

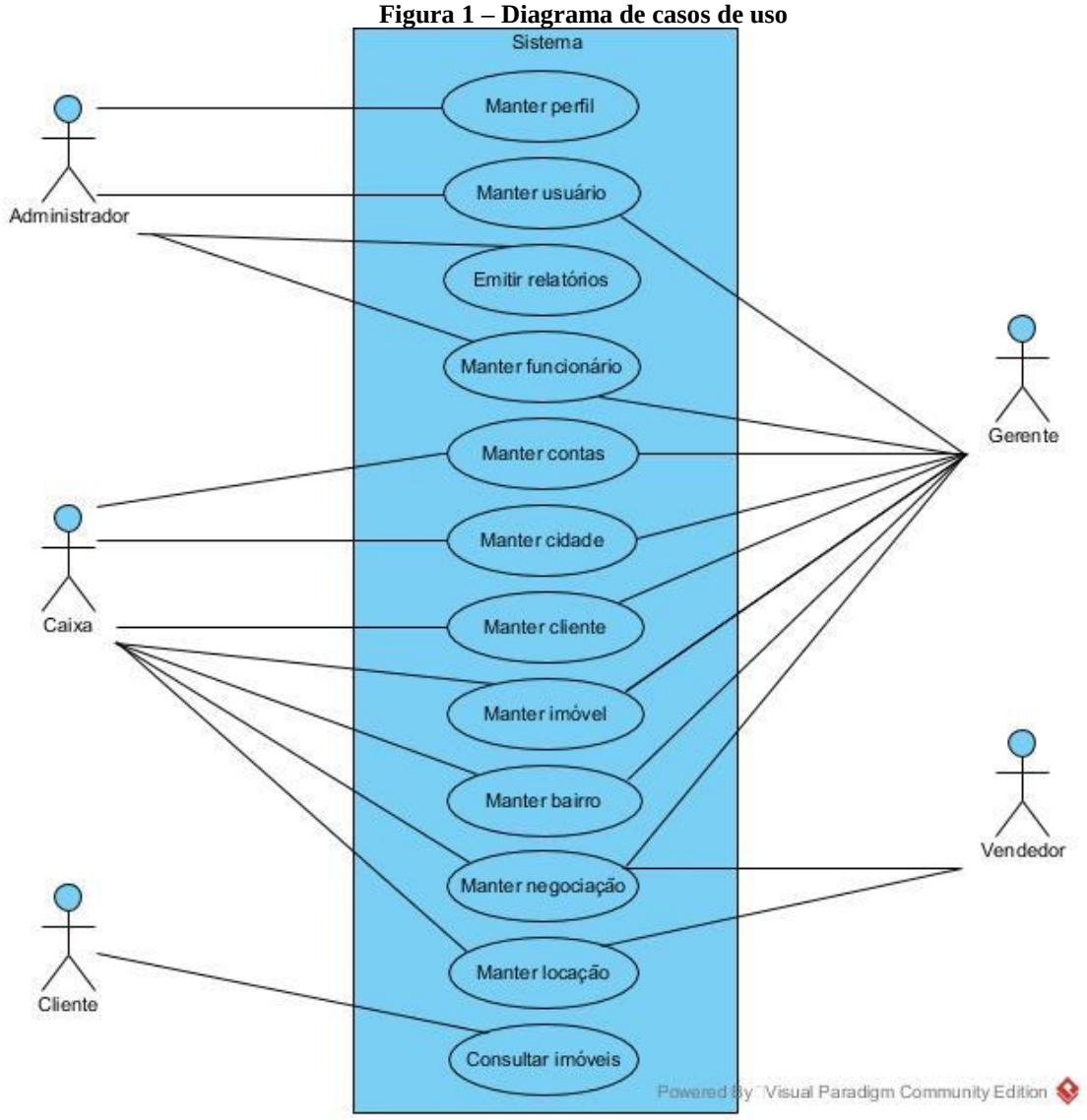
O sistema gera relatórios de compra e venda de imóveis, rescisões de contratos e contratação de locações, movimentação de caixa ambos relatórios filtrados por período, responsável pela negociação, entre outras informações.

O sistema conta com uma área que irá possibilitar aos clientes efetuarem buscas de imóveis disponíveis, podendo filtrar por determinadas cidades, bairros, quantidade de quartos ou vagas de garagem. Dentro das informações dos imóveis, é possível visualizar informações adicionais, como endereço, valor, se aceita trocas caso o imóvel esteja à venda, além de um mapa mostrando a localização.

4.2 MODELAGEM DO SISTEMA

O diagrama de casos de uso apresentado na Figura 1 contém as funcionalidades essenciais do sistema realizadas pelos seus atores que são: administrador, gerente, caixa, vendedor e cliente.

Figura 1 – Diagrama de casos de uso



Fonte: Autoria própria.

No Quadro 2 está a descrição da operação cadastrar (incluir) do caso de uso manter perfil.

Quadro 2 – Caso de uso manter perfil: operação cadastrar

Caso de uso: Manter perfil: operação cadastrar. **Descrição:**

Quando necessário, o administrador cadastrará um novo perfil de permissões.

Atores:

Administrador.

Pré-condição:

Estar autenticado no sistema.

Sequência de Eventos:

1. Administrador acessa a tela de listagem de perfis.
2. Administrador pressiona o botão novo para cadastrar perfil de permissões.
3. Sistema apresenta formulário de cadastro.
4. Administrador preenche e envia formulário.
5. Sistema informa que os dados foram incluídos no banco de dados.

Pós-Condição:

Dados do novo perfil de permissões inseridos no banco de dados.

Fonte: Autoria própria.

No Quadro 3 está a descrição da operação editar do caso de uso manter perfil.

Quadro 3 – Caso de uso manter perfil: operação editar

Caso de uso:

Manter perfil: operação editar.

Descrição:

Quando necessário, o administrador editará um perfil de permissões.

Atores:

Administrador.

Pré-condição:

Estar autenticado no sistema.

Sequência de Eventos:

1. Administrador acessa a tela de listagem de perfis.
2. Administrador pressiona o botão editar no perfil desejado.
3. Sistema apresenta formulário com campos de cadastro preenchidos com os dados do perfil de permissões selecionado.
4. Administrador edita os campos desejados e envia formulário para que seja persistido no banco de dados.
5. Sistema registra alterações no banco de dados e informa que os dados foram alterados.

Pós-Condição:

Dados do perfil de permissões editados no banco de dados.

Fonte: Autoria própria.

No Quadro 4 está a descrição da operação remover do caso de uso manter perfil.

Quadro 4 – Caso de uso manter perfil: operação remover

Caso de uso:
Manter perfil: operação remover.
Descrição:
Quando necessário, o administrador removerá de um perfil de permissões.
Atores:
Administrador.
Pré-condição:
Estar autenticado no sistema.
Sequência de Eventos:
1. Administrador acessa a tela de listagem de perfis.
2. Administrador pressiona o botão remover no perfil que deseja excluir.
3. Sistema retorna aviso que após excluído os registros, a ação não poderá ser desfeita.
4. Administrador confirma exclusão.
5. Sistema efetua a exclusão no banco de dados.
5. Sistema informa que os dados foram excluídos do banco de dados.
Pós-Condição:
Dados do perfil de permissões removidos no banco de dados.

Fonte: Autoria própria.

No Quadro 5 está a descrição da operação cadastrar (incluir) do caso de uso manter bairro.

Quadro 5 – Caso de uso manter bairro: operação cadastrar

Caso de uso:
Manter bairro: operação cadastrar.
Descrição:
Quando necessário, o gerente ou caixa cadastrará um novo bairro.
Atores:
Gerente ou caixa.
Pré-condição:
Estar autenticado no sistema.
Sequência de Eventos:
1. O ator acessa a tela de listagem de bairros.
2. O ator pressiona o botão novo para cadastrar o bairro.
3. Sistema apresenta formulário de cadastro.
4. O ator preenche e envia formulário.
5. Sistema informa que os dados foram incluídos no banco de dados.
Pós-Condição:
Dados do novo bairro inseridos no banco de dados.

Fonte: Autoria própria.

No Quadro 6 está a descrição da operação editar do caso de uso manter bairro.

Quadro 6 – Caso de uso manter bairro: operação editar

Caso de uso: Manter bairro: operação editar. **Descrição:** Quando necessário, o gerente ou caixa editará um bairro. **Atores:** Gerente ou caixa. **Pré-condição:** Estar autenticado no sistema. **Sequência de Eventos:** 1. O ator acessa a tela de listagem de bairros. 2. O ator pressiona o botão editar no bairro desejado. 3. Sistema apresenta formulário com campos de cadastro preenchidos com os dados do bairro selecionado. 4. O ator edita os campos desejados e envia formulário para que seja persistido no banco de dados. 5. Sistema registra alterações no banco de dados e informa que os dados foram alterados. **Pós-Condição:** Dados do bairro editados no banco de dados.

Fonte: Autoria própria.

No Quadro 7 está a descrição da operação remover do caso de uso manter bairro.

Quadro 7 – Caso de uso manter bairro: operação remover

Caso de uso: Manter bairro: operação remover. **Descrição:** Quando necessário, o gerente ou caixa removerá de um bairro. **Atores:** Gerente ou caixa. **Pré-condição:** Estar autenticado no sistema. **Sequência de Eventos:** 1. O ator acessa a tela de listagem de bairros. 2. O ator pressiona o botão remover no bairro que deseja excluir. 3. Sistema retorna aviso que após excluído os registros, a ação não poderá ser desfeita. 4. O ator confirma exclusão. 5. Sistema efetua a exclusão no banco de dados. 5. Sistema informa que os dados foram excluídos do banco de dados. **Pós-Condição:**

Dados do bairro removidos no banco de dados.

Fonte: Autoria própria.

No Quadro 8 está a descrição da operação cadastrar (incluir) do caso de uso manter usuário.

Quadro 8 – Caso de uso manter usuário: operação cadastrar

Caso de uso: Manter usuário: operação cadastrar. **Descrição:** Quando necessário, o administrador ou gerente cadastrará um novo usuário para utilização do sistema. **Atores:** Administrador ou gerente. **Pré-condição:** Estar autenticado no sistema. **Sequência de Eventos:** 1. O ator acessa a tela de listagem de usuários. 2. O ator pressiona o botão novo para cadastrar o usuário. 3. Sistema apresenta formulário de cadastro. 4. O ator preenche e envia formulário. 5. Sistema informa que os dados foram incluídos no banco de dados. **Pós-Condição:** Dados do novo usuário inseridos no banco de dados.

Fonte: Autoria própria.

No Quadro 9 está a descrição da operação editar do caso de uso manter usuário.

Quadro 9 – Caso de uso manter usuário: operação editar

Caso de uso: Manter usuário: operação editar. **Descrição:** Quando necessário, o administrador ou gerente editará um usuário. **Atores:** Administrador ou gerente. **Pré-condição:** Estar autenticado no sistema. **Sequência de Eventos:** 1. O ator acessa a tela de listagem de usuários. 2. O ator pressiona o botão editar no usuário desejado.

3. Sistema apresenta formulário com campos de cadastro preenchidos com os dados do usuário selecionado. 4. O ator edita os campos desejados e envia formulário para que seja persistido no banco de dados. 5. Sistema registra alterações no banco de dados e informa que os dados foram alterados. **Pós-Condição:** Dados do usuário editados no banco de dados.

Fonte: Autoria própria.

No Quadro 10 está a descrição da operação remover do caso de uso manter usuário.

Quadro 10 – Caso de uso manter usuário: operação remover

Caso de uso: Manter usuário: operação remover. **Descrição:** Quando necessário, o administrador ou gerente removerá um usuário. **Atores:** Administrador ou gerente. **Pré-condição:** Estar autenticado no sistema. **Sequência de Eventos:** 1. O ator acessa a tela de listagem de usuários. 2. O ator pressiona o botão remover no usuário que deseja excluir. 3. Sistema retorna aviso que após excluído os registros, a ação não poderá ser desfeita. 4. O ator confirma exclusão. 5. Sistema efetua a exclusão no banco de dados. 5. Sistema informa que os dados foram excluídos do banco de dados. **Pós-Condição:** Dados do usuário removidos no banco de dados.

Fonte: Autoria própria.

No Quadro 11 está a descrição da operação cadastrar (incluir) do caso de uso manter cliente.

Quadro 11 – Caso de uso manter cliente: operação cadastrar

Caso de uso: Manter cliente: operação cadastrar. **Descrição:** Quando necessário, o gerente ou caixa cadastrará um novo cliente. **Atores:** Gerente ou caixa.

Pré-condição:
Estar autenticado no sistema.
Sequência de Eventos:
1. O ator acessa a tela de listagem de cliente.
2. O ator pressiona o botão novo para cadastrar o cliente.
3. Sistema apresenta formulário de cadastro.
4. O ator preenche e envia formulário.
5. Sistema informa que os dados foram incluídos no banco de dados.
Pós-Condição:
Dados do novo cliente inseridos no banco de dados.

Fonte: Autoria própria.

No Quadro 12 está a descrição da operação editar do caso de uso manter cliente.

Quadro 12 – Caso de uso manter cliente: operação editar

Caso de uso:
Manter cliente: operação editar.
Descrição:
Quando necessário, o gerente ou caixa editará um usuário.
Atores:
Gerente ou caixa.
Pré-condição:
Estar autenticado no sistema.
Sequência de Eventos:
1. O ator acessa a tela de listagem de clientes.
2. O ator pressiona o botão editar no cliente desejado.
3. Sistema apresenta formulário com campos de cadastro preenchidos com os dados do cliente selecionado.
4. O ator edita os campos desejados e envia formulário para que seja persistido no banco de dados.
5. Sistema registra alterações no banco de dados e informa que os dados foram alterados.
Pós-Condição:
Dados do cliente editados no banco de dados.

Fonte: Autoria própria.

No Quadro 13 está a descrição da operação cadastrar (incluir) do caso de uso manter funcionário.

Quadro 13 – Caso de uso manter funcionário: operação cadastrar

Caso de uso:

Manter funcionário: operação cadastrar.

Descrição:

Quando necessário, administrador ou gerente cadastrará um novo funcionário.

Atores:

Administrador ou gerente.

Pré-condição:

Estar autenticado no sistema.

Sequência de Eventos:

1. O ator acessa a tela de listagem de funcionários.
2. O ator pressiona o botão novo para cadastrar o funcionário.
3. Sistema apresenta formulário de cadastro.
4. O ator preenche e envia formulário.
5. Sistema informa que os dados foram incluídos no banco de dados.

Pós-Condição:

Dados do novo funcionário inseridos no banco de dados.

Fonte: Autoria própria.

No Quadro 14 está a descrição da operação editar do caso de uso manter funcionário.

Quadro 14 – Caso de uso manter funcionário: operação editar

Caso de uso:

Manter funcionário: operação editar.

Descrição:

Quando necessário, administrador ou gerente editará um funcionário.

Atores:

Administrador ou gerente.

Pré-condição:

Estar autenticado no sistema.

Sequência de Eventos:

1. O ator acessa a tela de listagem de funcionário.
2. O ator pressiona o botão editar no funcionário desejado.
3. Sistema apresenta formulário com campos de cadastro preenchidos com os dados do cliente selecionado.
4. O ator edita os campos desejados e envia formulário para que seja persistido no banco de dados.
5. Sistema registra alterações no banco de dados e informa que os dados foram alterados.

Pós-Condição:

Dados do funcionário editados no banco de dados.

Fonte: Autoria própria.

No Quadro 15 está a descrição da operação cadastrar (incluir) do caso de uso manter imóvel.

Quadro 15 – Caso de uso manter imóvel: operação cadastrar

Caso de uso: Manter imóvel: operação cadastrar. **Descrição:** Quando necessário, gerente ou caixa cadastrará um novo imóvel. **Atores:** Gerente ou caixa. **Pré-condição:** Estar autenticado no sistema. **Sequência de Eventos:** 1. O ator acessa a tela de listagem de imóveis. 2. O ator pressiona o botão novo para cadastrar o imóvel. 3. Sistema apresenta formulário de cadastro. 4. O ator preenche e envia formulário. 5. Sistema informa que os dados foram incluídos no banco de dados. **Pós-Condição:** Dados do novo imóvel inseridos no banco de dados.

Fonte: Autoria própria.

No Quadro 16 está a descrição da operação editar do caso de uso manter imóvel.

Quadro 16 – Caso de uso manter imóvel: operação editar

Caso de uso: Manter imóvel: operação editar. **Descrição:** Quando necessário, o gerente ou caixa editará um imóvel. **Atores:** Gerente ou caixa. **Pré-condição:** Estar autenticado no sistema. **Sequência de Eventos:** 1. O ator acessa a tela de listagem de imóveis. 2. O ator pressiona o botão editar no imóvel desejado. 3. Sistema apresenta formulário com campos de cadastro preenchidos com os dados do imóvel selecionado. 4. O ator edita os campos desejados e envia formulário para que seja persistido no banco de dados.

5. Sistema registra alterações no banco de dados e informa que os dados foram alterados. **Pós-Condição:** Dados do imóvel editados no banco de dados.

Fonte: Autoria própria.

No Quadro 17 está a descrição da operação cadastrar (incluir) do caso de uso manter negociação.

Quadro 17 – Caso de uso manter negociação: operação cadastrar

Caso de uso: Manter negociação: operação cadastrar. **Descrição:** Quando necessário, gerente ou vendedor cadastrará uma nova negociação de venda ou aluguel de um imóvel. **Atores:** Gerente ou vendedor. **Pré-condição:** Estar autenticado no sistema. **Sequência de Eventos:** 1. O ator acessa a tela de listagem de negociações. 2. O ator pressiona o botão novo para cadastrar a nova negociação de venda ou aluguel de um imóvel. 3. Sistema apresenta formulário de cadastro. 4. O ator preenche e envia formulário. 5. Sistema informa que os dados foram incluídos no banco de dados. **Pós-Condição:** Dados da negociação do imóvel inseridos no banco de dados.

Fonte: Autoria própria.

Quadro 18 - Caso de uso manter negociação: operação editar

Caso de uso: Manter negociação: operação editar. **Descrição:** Quando necessário, gerente ou vendedor editará uma nova negociação de venda ou de aluguel de um imóvel. **Atores:** Gerente ou vendedor. **Pré-condição:** Estar autenticado no sistema. **Sequência de Eventos:** 1. O ator acessa a tela de listagem de negociações.

2. O ator pressiona o botão editar na negociação desejada.

3. Sistema apresenta formulário com campos de cadastro preenchidos com os dados da negociação selecionada.

4. O ator edita os campos desejados e envia formulário para que seja persistido no banco de dados.

5. Sistema registra alterações no banco de dados e informa que os dados foram alterados.

Pós-Condição:

Dados da negociação do imóvel alterados no banco de dados.

Fonte: Autoria própria.

Quadro 19 – Caso de uso manter negociação: operação finalizar

Caso de uso:

Manter negociação: operação finalizar.

Descrição:

Quando necessário, gerente ou vendedor finalizará uma negociação de venda ou aluguel de um imóvel.

Atores:

Gerente ou vendedor.

Pré-condição:

Estar autenticado no sistema.

Sequência de Eventos:

1. O ator acessa a tela de listagem de negociações.

2. O ator pressiona o botão editar na negociação desejada.

3. Sistema apresenta formulário com campos de cadastro preenchidos com os dados da negociação selecionada.

4. O ator edita os campos desejados e envia formulário para que seja persistido no banco de dados.

5. Sistema gera contas a receber referente a aquela negociação, altera o status da negociação para finalizada, caso seja uma venda cria um novo imóvel vinculado ao comprador, e inativa o imóvel vinculado ao vendedor.

6. Sistema informa que os dados foram incluídos no banco de dados.

Pós-Condição:

Dados da negociação do imóvel inseridos no banco de dados.

Fonte: Autoria própria.

No Quadro 20 está a descrição da operação receber comissão do caso de uso manter negociação.

Quadro 20 – Caso de uso manter contas: operação quitar conta

Caso de uso:

Manter contas: operação quitar conta.

Descrição:

Quando necessário, o caixa quitará uma conta de negociação de um imóvel.

Atores:

Caixa.

Pré-condição:

Estar autenticado no sistema.

Sequência de Eventos:

1. O caixa acessa a tela de listagem de contas a pagar ou receber.
2. O caixa busca a conta pelo nome do comprador ou vendedor.
3. O caixa pressiona o botão editar para listar as informações da conta.
4. Sistema apresenta formulário com campos de cadastro preenchidos com os dados do da conta selecionado.
5. O caixa informa a data de pagamento e confirma o pagamento.
6. O sistema altera o status da parcela.
7. Sistema informa que os dados foram incluídos no banco de dados.

Pós-Condição:

Dados da quitação da conta alterados no banco de dados.

Fonte: Autoria própria.

No Quadro 21 está a descrição da operação cadastrar (incluir) do caso de uso manter cidade.

Quadro 21 – Caso de uso manter cidade: operação cadastrar

Caso de uso:

Manter cidade: operação cadastrar.

Descrição:

Quando necessário, o gerente ou caixa cadastrará uma nova cidade.

Atores:

Gerente ou caixa.

Pré-condição:

Estar autenticado no sistema.

Sequência de Eventos:

1. O ator acessa a tela de listagem de cidades.
2. O ator pressiona o botão novo para cadastrar a cidade.
3. Sistema apresenta formulário de cadastro.
4. O ator preenche e envia formulário.
5. Sistema informa que os dados foram incluídos no banco de dados.

Pós-Condição:

Dados da nova cidade inseridos no banco de dados.

Fonte: Autoria própria.

No Quadro 22 está a descrição da operação editar do caso de uso manter cidade.

Quadro 22 – Caso de uso manter cidade: operação editar

Caso de uso: Manter cidade: operação editar. **Descrição:** Quando necessário, o gerente ou caixa editará uma cidade. **Atores:** Gerente ou caixa. **Pré-condição:** Estar autenticado no sistema. **Sequência de Eventos:** 1. O ator acessa a tela de listagem de cidades. 2. O ator pressiona o botão editar na cidade desejada. 3. Sistema apresenta formulário com campos de cadastro preenchidos com os dados da cidade selecionada. 4. O ator edita os campos desejados e envia formulário para que seja persistido no banco de dados. 5. Sistema registra alterações no banco de dados e informa que os dados foram alterados. **Pós-Condição:** Dados da cidade editados no banco de dados.

Fonte: Autoria própria.

No Quadro 23 está a descrição da operação remover do caso de uso manter cidade.

Quadro 23 – Caso de uso manter cidade: operação remover

Caso de uso: Manter cidade: operação remover. **Descrição:** Quando necessário, o gerente ou caixa removerá de uma cidade. **Atores:** Gerente ou caixa. **Pré-condição:** Estar autenticado no sistema. **Sequência de Eventos:** 1. O ator acessa a tela de listagem de cidades. 2. O ator pressiona o botão remover na cidade que deseja excluir. 3. Sistema retorna aviso que após excluído os registros, a ação não poderá ser desfeita. 4. O ator confirma exclusão. 5. Sistema efetua a exclusão no banco de dados. 5. Sistema informa que os dados foram excluídos do banco de dados. **Pós-Condição:** Dados da cidade removidos no banco de dados.

Fonte: Autoria própria.

No Quadro 24 está a descrição da operação buscar do caso de uso consultar imóveis.

Quadro 24 – Caso de uso consultar imóveis: operação buscar

Caso de uso: Consultar imóveis: operação buscar. **Descrição:** Quando necessário, o cliente irá buscar um imóvel. **Atores:** Cliente. **Pré-condição:** Estar com o site acessado. **Sequência de Eventos:** 1. O cliente acessa o site de imóveis. 2. O cliente seleciona especificações do imóvel. 3. O cliente pressiona o botão buscar para filtrar os imóveis desejados. 4. Sistema apresenta listagem de imóveis de acordo com as especificações selecionadas pelo cliente. 5. O cliente pressiona em cima do imóvel para visualizar informações adicionais sobre o imóvel. **Pós-Condição:** Lista de imóveis selecionada.

Fonte: Autoria própria.

No Quadro 25 está a descrição do caso de uso emitir relatórios.

Quadro 25 – Caso de uso emitir relatórios

Caso de uso: Emitir relatórios. **Descrição:** Quando necessário, o administrador irá emitir relatórios. **Atores:** Administrador. **Pré-condição:** Estar autenticado no sistema. **Sequência de Eventos:** 1. O administrador acessa a tela de relatórios. 2. O administrador seleciona os parâmetros de busca. 3. O administrador pressiona o botão gerar para emitir o relatório. 4. Sistema apresenta o relatório de acordo com os parâmetros selecionados pelo gerente. **Pós-Condição:**

Emissão do relatório.

Fonte: Autoria própria.

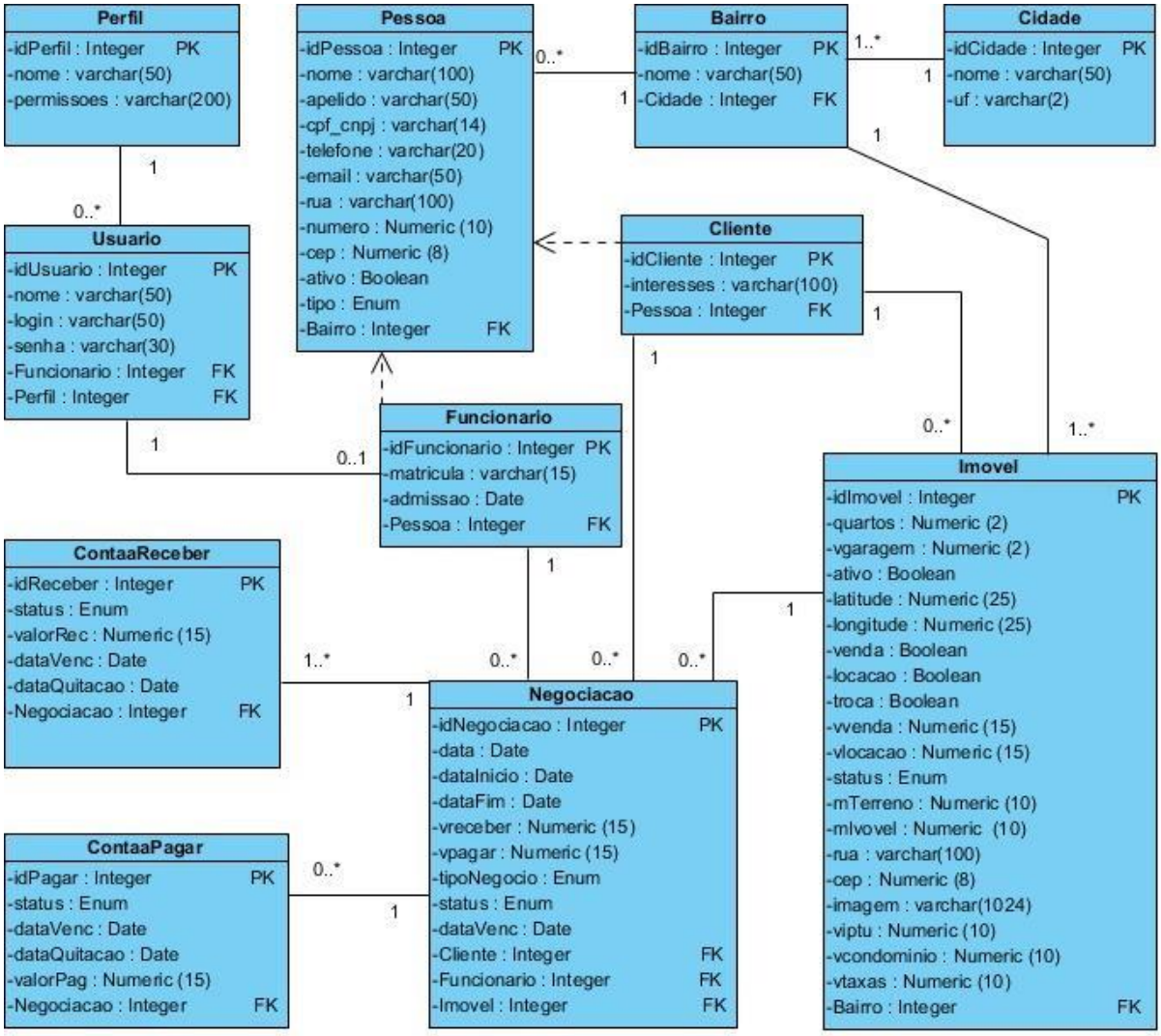
A Figura 2 representa o Diagrama de Entidades e Relacionamentos (DER) do banco de dados utilizando a *PostgreSQL*. Os campos de chave primária são do tipo serial para que sejam gerados automaticamente pelo banco de dados.

A tabela de permissões dos usuários foi gerada por meio de relacionamento de muitos para muitos entre usuário e permissões. Um usuário pode possuir uma ou muitas permissões, estando sempre relacionado a um perfil de permissões.

As tabelas cliente e funcionário irão herdar os atributos da tabela pessoa.

A tabela negociação será utilizada para as vendas e locações, tendo sempre vinculado a negociação um funcionário, um cliente comprador/locatário e um cliente vendedor/locador. Caso a negociação seja uma locação irá possuir uma ou mais conta a receber/pagar, em uma venda, irá possuir somente uma ou mais conta a receber, referente ao valor da comissão da negociação.

Figura 2 - Diagrama de entidades e relacionamentos



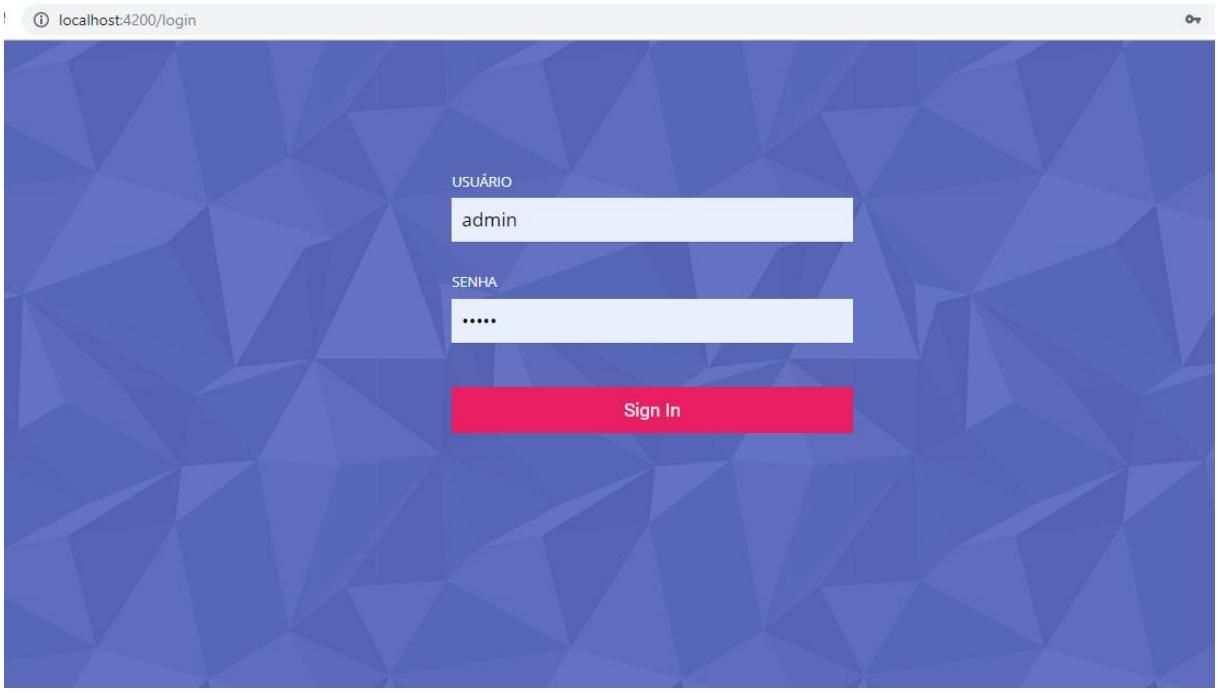
Fonte: Autoria própria.

4.3 APRESENTAÇÃO DO SISTEMA

Todas as telas do sistema compartilham duas partes importantes para a padronização do leiaute que são os menus laterais e páginas de cadastros. O administrador do sistema tem a função de prover e gerenciar as informações utilizadas no sistema.

O acesso ao sistema web é feito por meio do nome de usuário e senha. A Figura 3 apresenta a tela inicial do sistema web que o usuário utiliza para efetuar a autenticação, essa tela é composta apenas pelos campos usuário e senha.

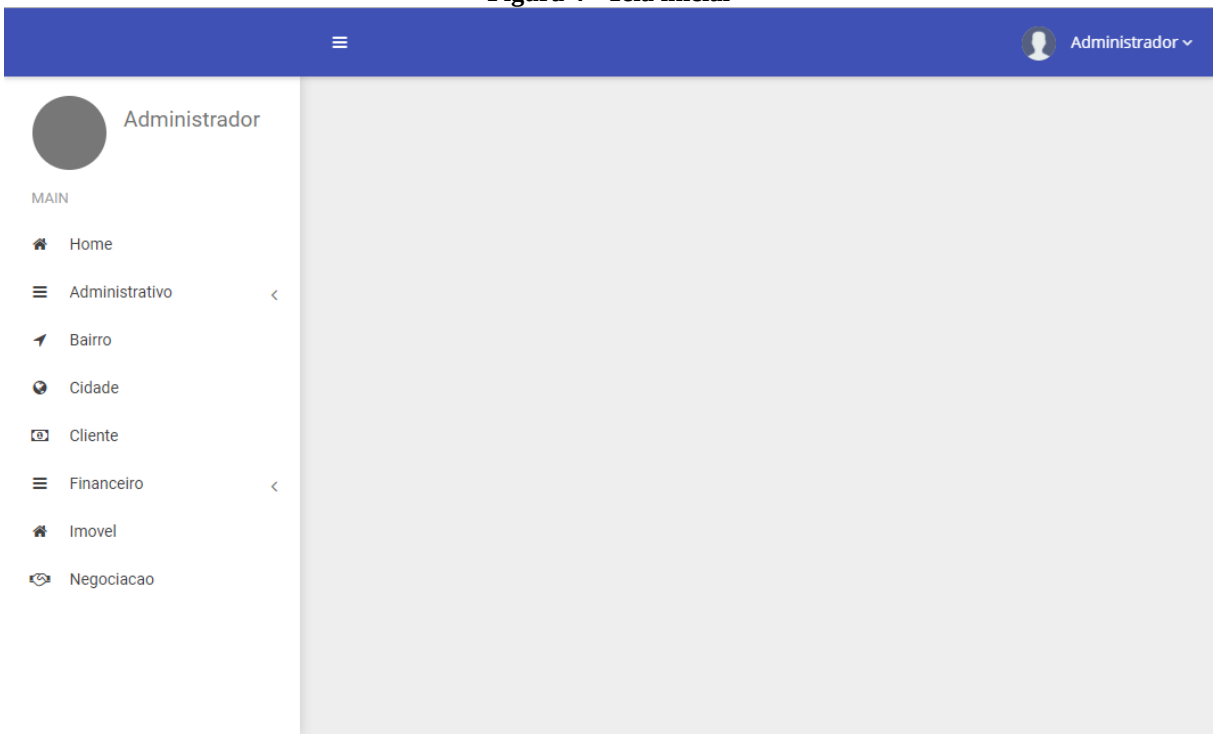
Figura 3 - Tela de *login*



Fonte: Autoria própria.

A Figura 4 apresenta a tela inicial do sistema, nela pode ser visto todos os menus e sub-menus para acesso aos diferentes módulos do sistema.

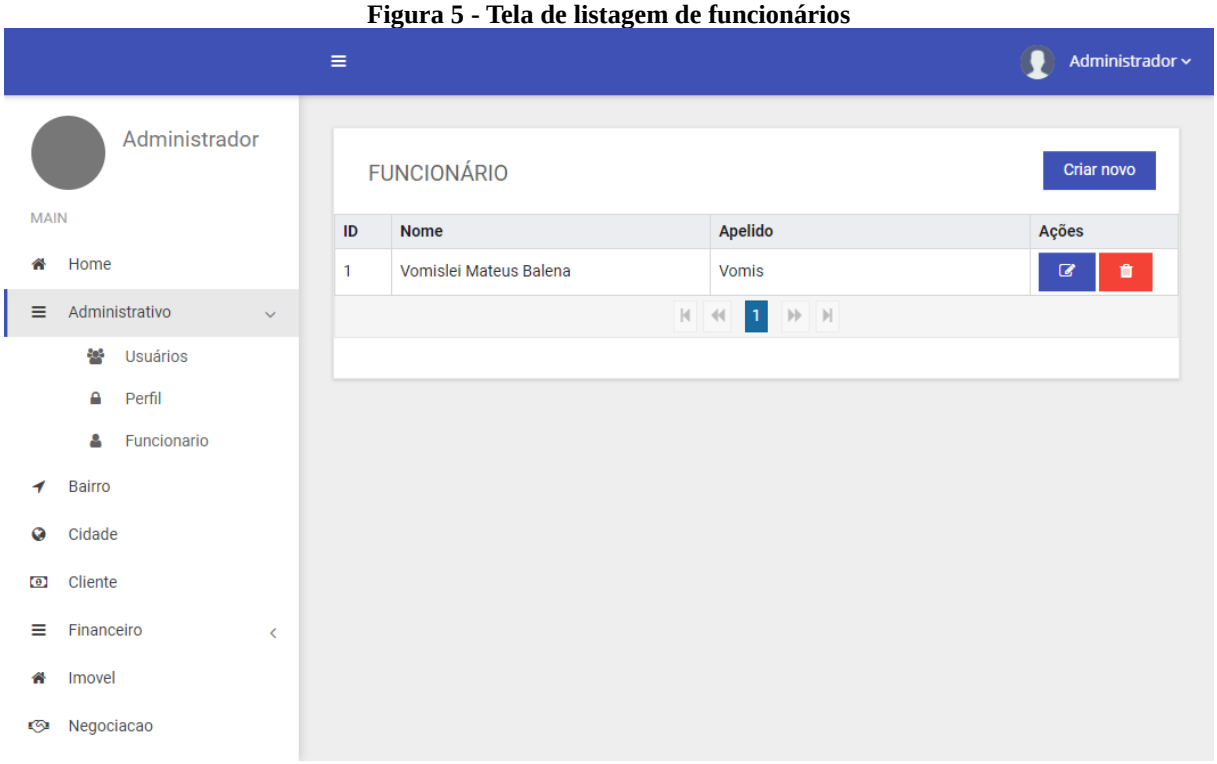
Figura 4 - Tela inicial



Fonte: Autoria própria.

A Figura 5 apresenta a tela de listagem de funcionários. Todos os cadastros do sistema seguem esse mesmo modelo. Na tabela são listadas todas as informações cadastradas com as opções de clicar no item para editar ou remover, ou em “criar novo” para cadastrar um novo item. O componente utilizado para listar os dados foi o *p-table* da biblioteca *DataTable*.

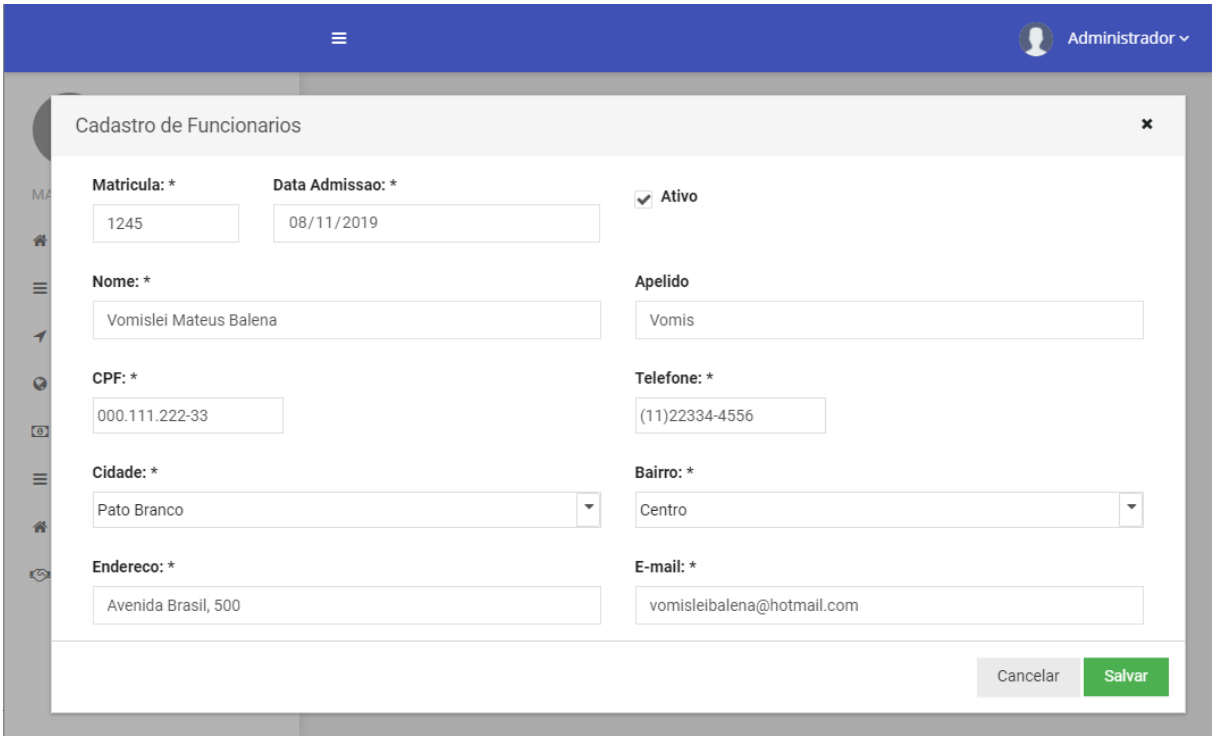
Figura 5 - Tela de listagem de funcionários



Fonte: Autoria própria.

Clicando em um item da listagem é acessada a tela de edição. A Figura 6 apresenta a tela de cadastro de funcionários. Por meio dessa tela o administrador pode editar as informações do funcionário. Os campos de texto são elementos HTML do tipo *input text*. O campo de data também se trata de um elemento HTML porém do tipo *date*. Para os *checkbox* é utilizado o elemento p-*checkbox*, para as máscaras dos campos o p-*inputMask*, e para o campo de seleção de cidades, clientes entre outros, foi utilizado o p-*dropdown*, todos esses componentes fazem parte da biblioteca *PrimeNg*.

Figura 6 - Tela de cadastro de funcionário

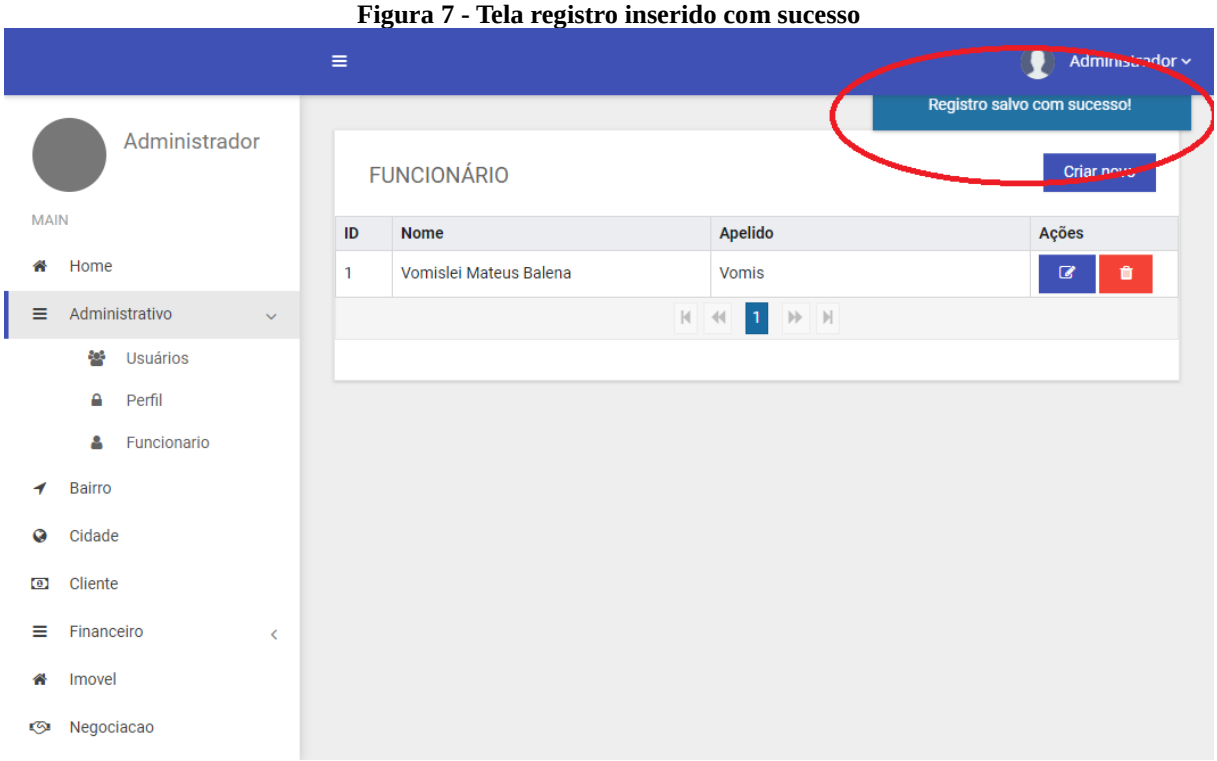


Fonte: Autoria própria.

As telas de cadastro são utilizadas tanto para a inclusão quanto para a edição das informações, e as informações são gravadas pelo botão salvar.

O resultado da operação é apresentado ao usuário por meio de mensagens no canto direito superior da tela. A Figura 7 exemplifica uma mensagem de resposta ao usuário que indica que a operação foi realizada com sucesso. Após a inclusão das informações, o sistema redireciona para a tela de listagem do item que foi cadastrado.

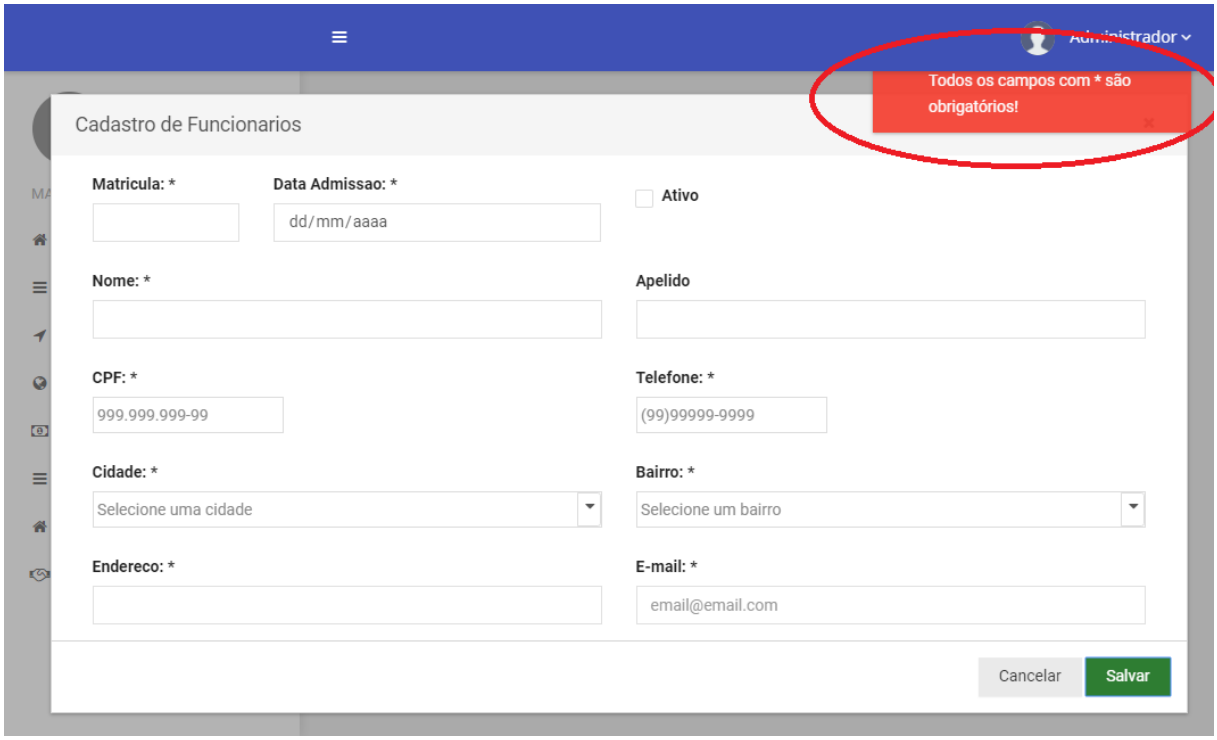
Figura 7 - Tela registro inserido com sucesso



Fonte: Autoria própria.

Na Figura 8 está um exemplo de mensagem de erro. No caso de falta de informação ou inconsistência dos dados, o sistema apresenta uma mensagem de erro, neste exemplo o cadastro de funcionário não possui a informação de bairro.

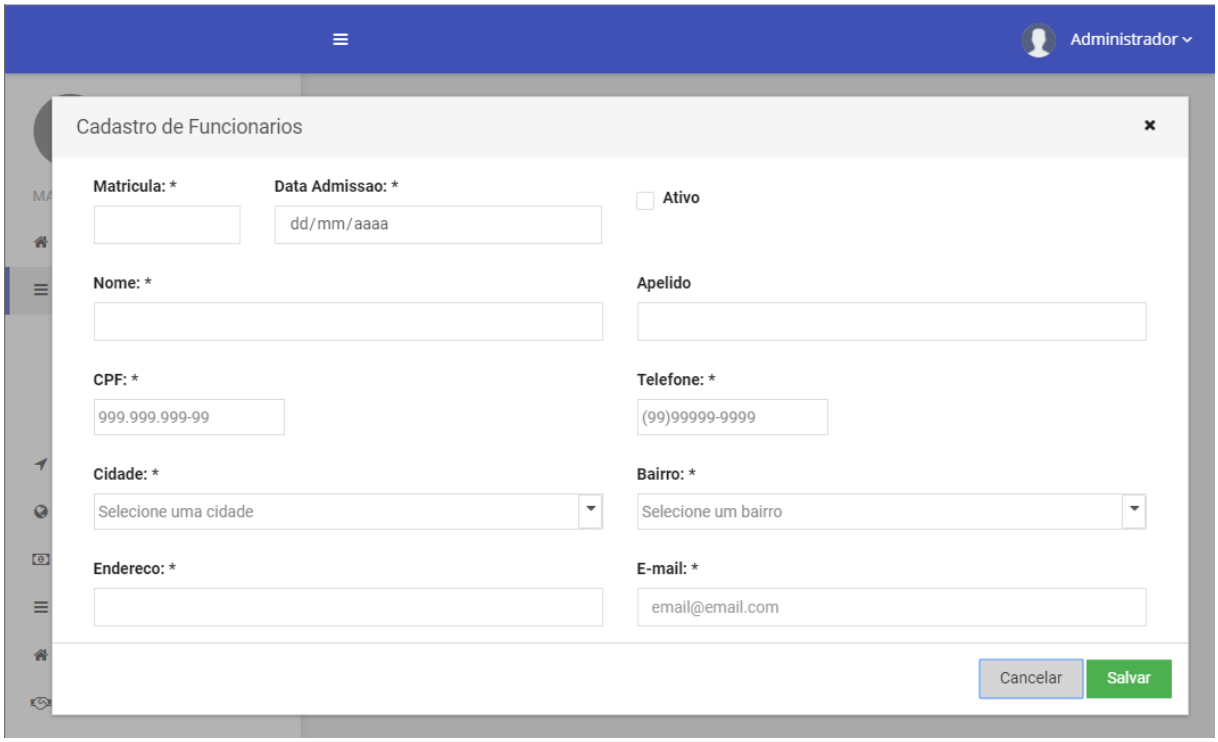
Figura 8 - Tela de erro ao inserir registro



Fonte: Autoria própria.

A Figura 9 apresenta a tela de cadastro de funcionários, nela é possível inserir todos os dados relacionados ao cadastro de um novo funcionário, nela é listado todas as cidades e seus respectivos bairros, a tela possui verificação de campos obrigatórios, os quais estão marcados com asterisco (*).

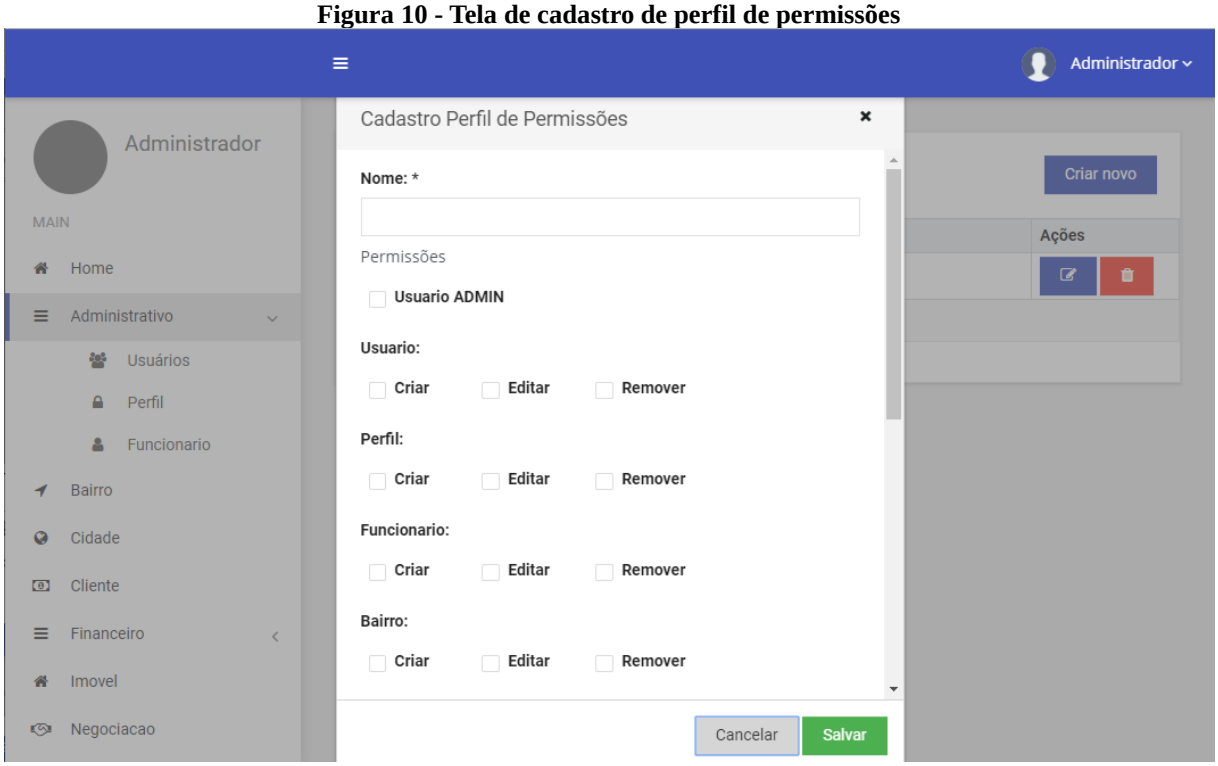
Figura 9 - Tela de cadastro de funcionários



Fonte: Autoria própria.

A Figura 10 apresenta a tela de cadastro de perfil de permissões, as permissões definem o que o usuário terá ou não acesso, como por exemplo, criar, editar e até remover registros. O usuário com a permissão ADMIN tem acesso completo a todos os módulos do sistema por padrão.

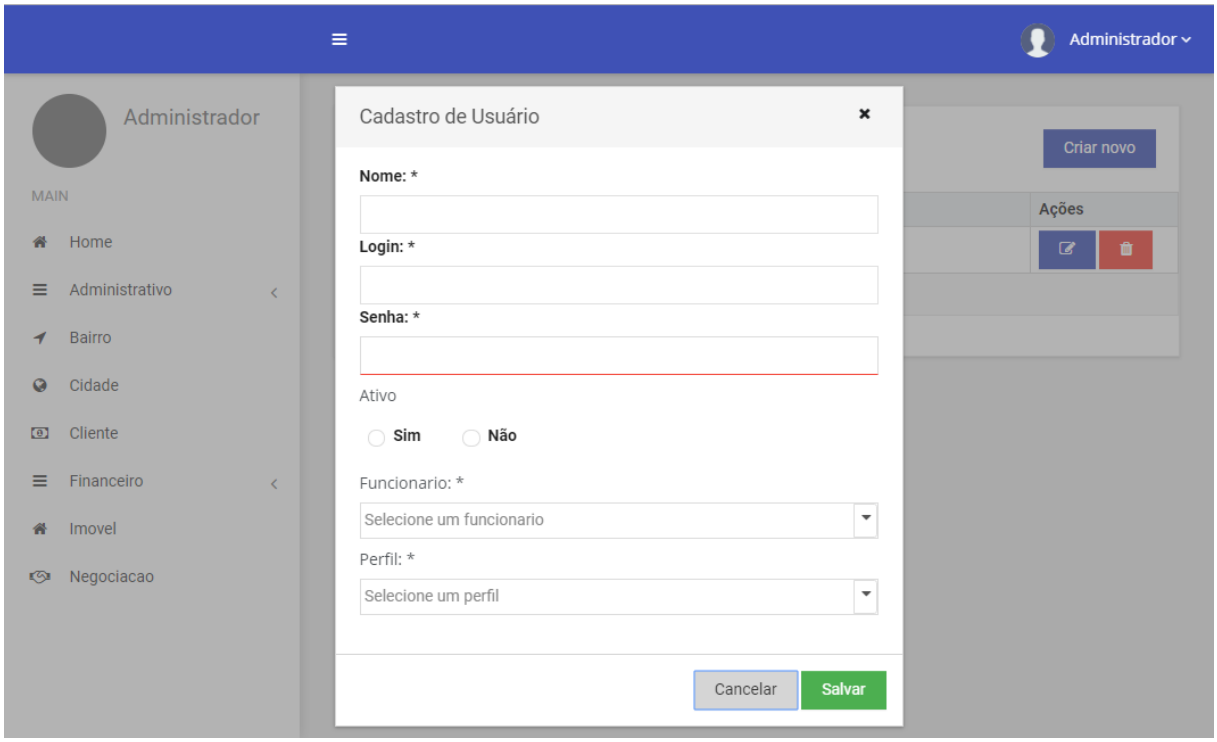
Figura 10 - Tela de cadastro de perfil de permissões



Fonte: Autoria própria.

A Figura 11 apresenta a tela de cadastro de usuário do sistema, sendo assim, ele somente poderá ser criado após o cadastro do funcionário já ter sido efetuado conforme apresentado na Figura 6.

Figura 11 - Tela de cadastro de usuário

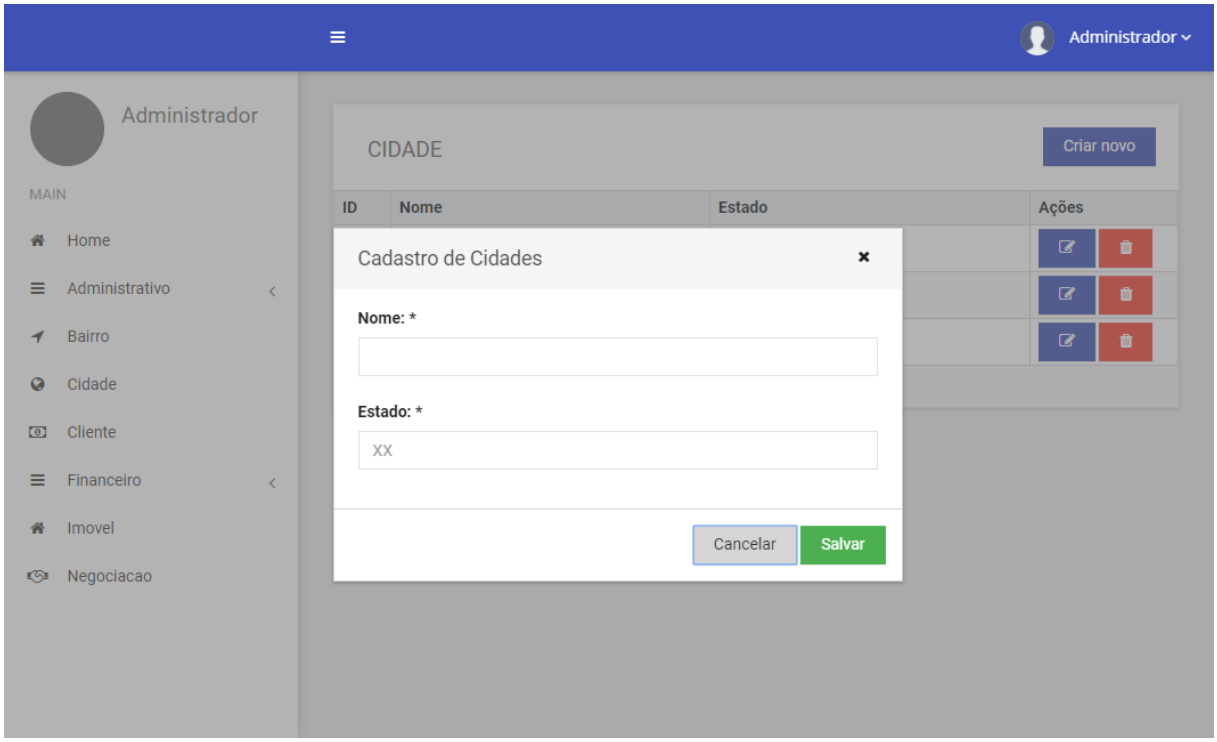


Fonte: Autoria própria.

Após todos os dados serem cadastrados pelo responsável, o usuário pode efetuar a autenticação no sistema utilizando seu nome de usuário e senha, por meio da tela apresentada na Figura 3. Assim que o usuário se autenticar, será apresentado somente os menus e funcionalidades de acordo com seu perfil de permissões. Sem a autenticação o usuário não pode acessar nenhuma tela da aplicação, sendo redirecionado para a tela de autenticação.

Para cadastrar uma nova cidade, a tela de cadastro de cidades é composta pelos campos Nome, e Estado, o qual é inserido somente a sigla, a tela possui verificação de todos os campos. A Figura 12 mostra a tela de cadastro de cidades.

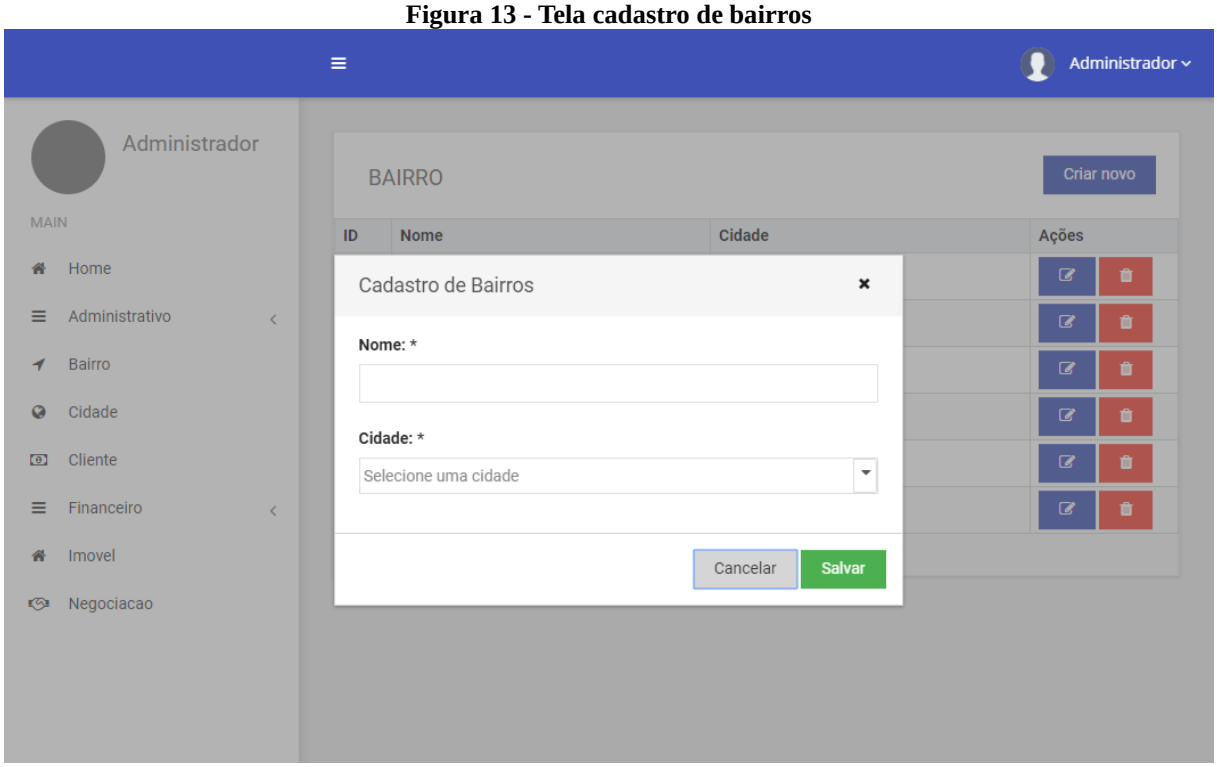
Figura 12 - Tela cadastro cidades



Fonte: Autoria própria.

Para cadastrar um novo bairro, a tela de cadastro de bairros é composta pelos campos Nome, e Cidade, a tela possui verificação de todos os campos. A Figura 13 mostra a tela de cadastro de bairros.

Figura 13 - Tela cadastro de bairros



Fonte: Autoria própria.

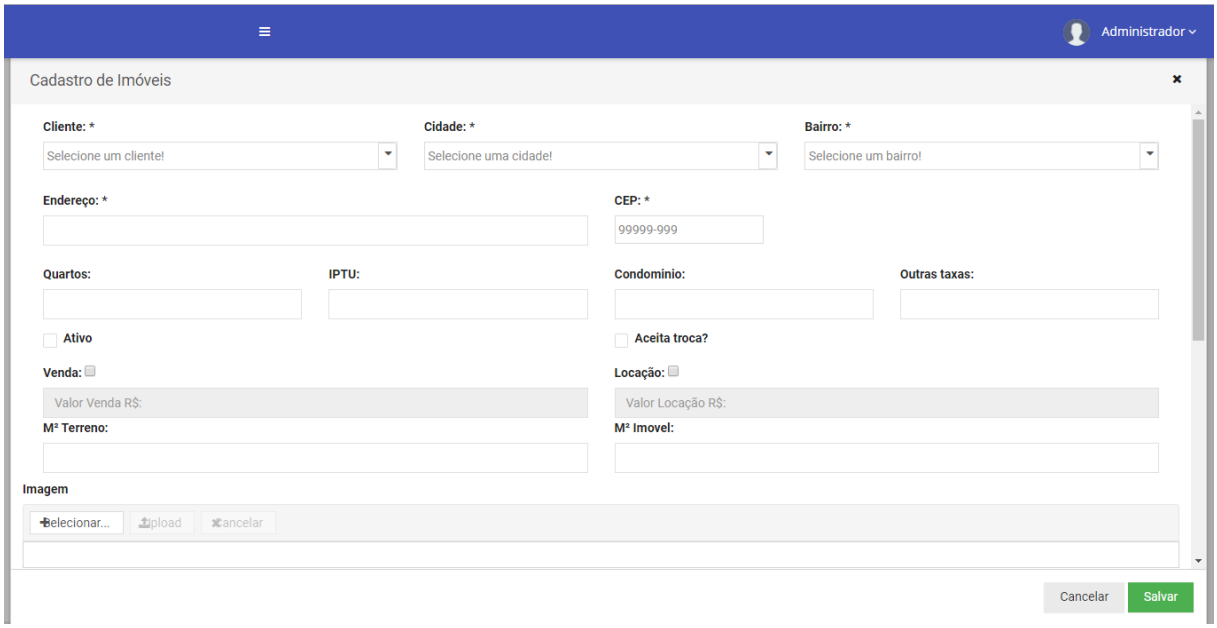
A Figura 14 apresenta a tela de cadastro de cliente, nela são listadas todas as cidades e seus respectivos bairros, a tela possui verificação nos campos obrigatórios.

Figura 14 - Tela de cadastro de cliente

Fonte: Autoria própria.

A Figura 15 e a Figura 16 apresentam a tela de cadastro de imóvel, somente os campos Cliente, Cidade, Bairro, Endereço e CEP são obrigatórios, os demais podem ou não ser preenchidos, de acordo com a necessidade do usuário. Por meio da tela de cadastro de imóvel também é possível inserir arquivos de imagem para complementar o cadastro do imóvel, bem como sua localização utilizando um mapa.

Figura 15 - Tela de cadastro de imóvel



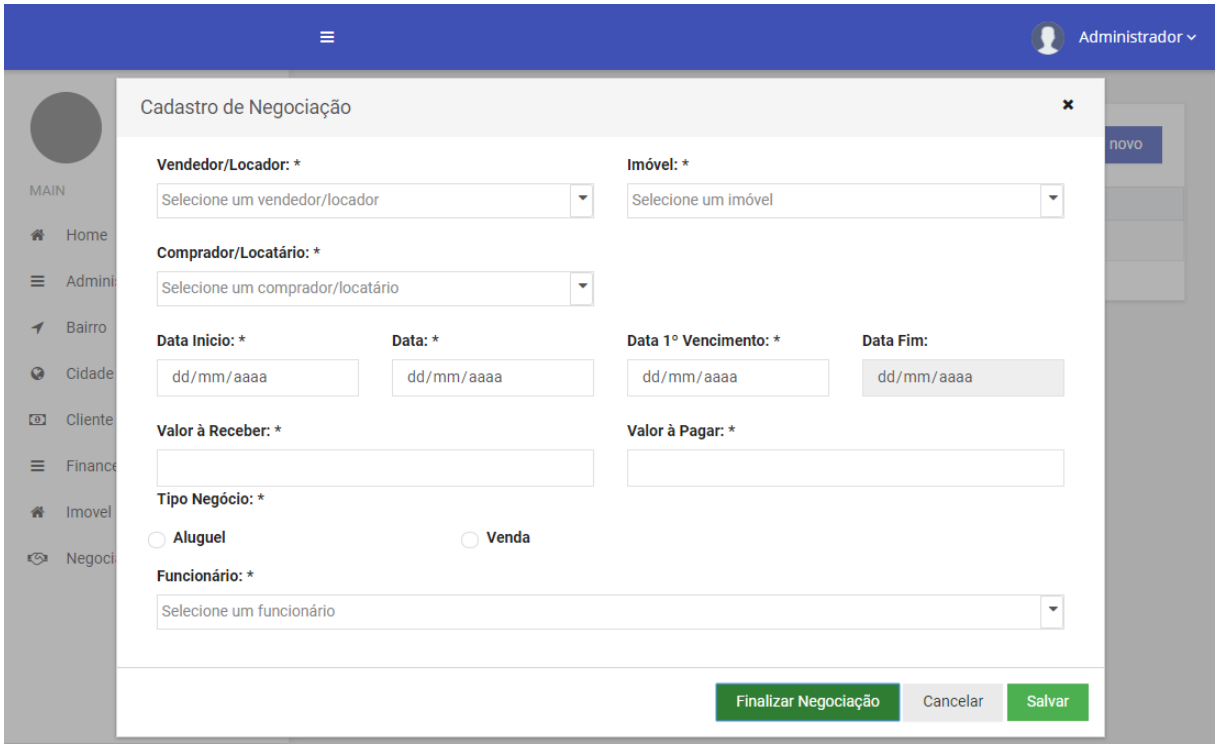
Fonte: Autoria própria.

Figura 16 - Tela de cadastro de imóvel

Fonte: Autoria própria.

A Figura 17 apresenta a tela de cadastro de negociação de um imóvel, essa tela traz a lista de clientes e seus respectivos imóveis ativos, além de possuir também outras informações complementares para a negociação.

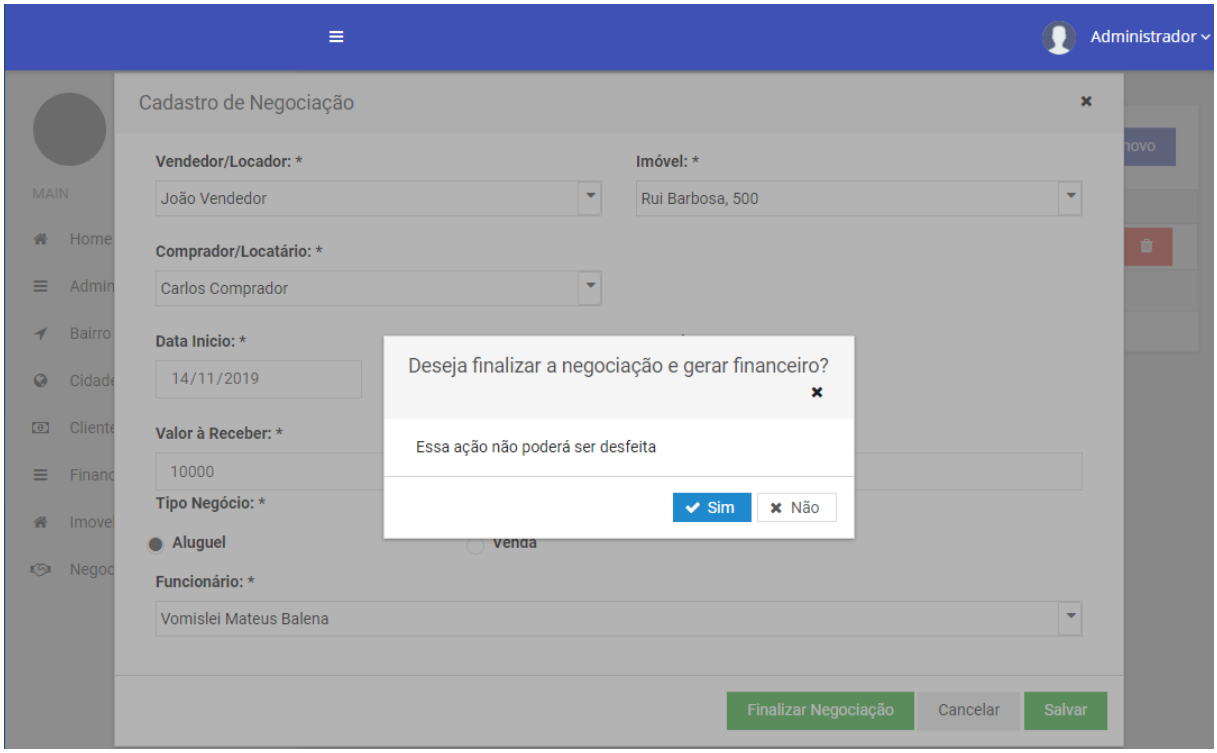
Figura 17 - Tela de cadastro de negociação



Fonte: Autoria própria.

A negociação pode ser somente salva utilizando o botão salvar, sendo que a situação irá ficar como “aberta”. O campo Data Fim não pode ser alterado pelo usuário, sendo preenchido somente na finalização da negociação após o usuário clicar na opção finalizar negociação. Dessa maneira a situação da negociação assumirá o valor “finalizada”. Automaticamente o sistema irá gerar parcelas de contas a pagar e a receber caso a negociação seja aluguel, ou somente a conta a receber caso seja venda. Caso a negociação seja do tipo venda, e possua uma negociação de aluguel com contas em aberto vinculadas ao imóvel, as mesmas serão canceladas. A Figura 18 apresenta a tela de confirmação de finalização da negociação.

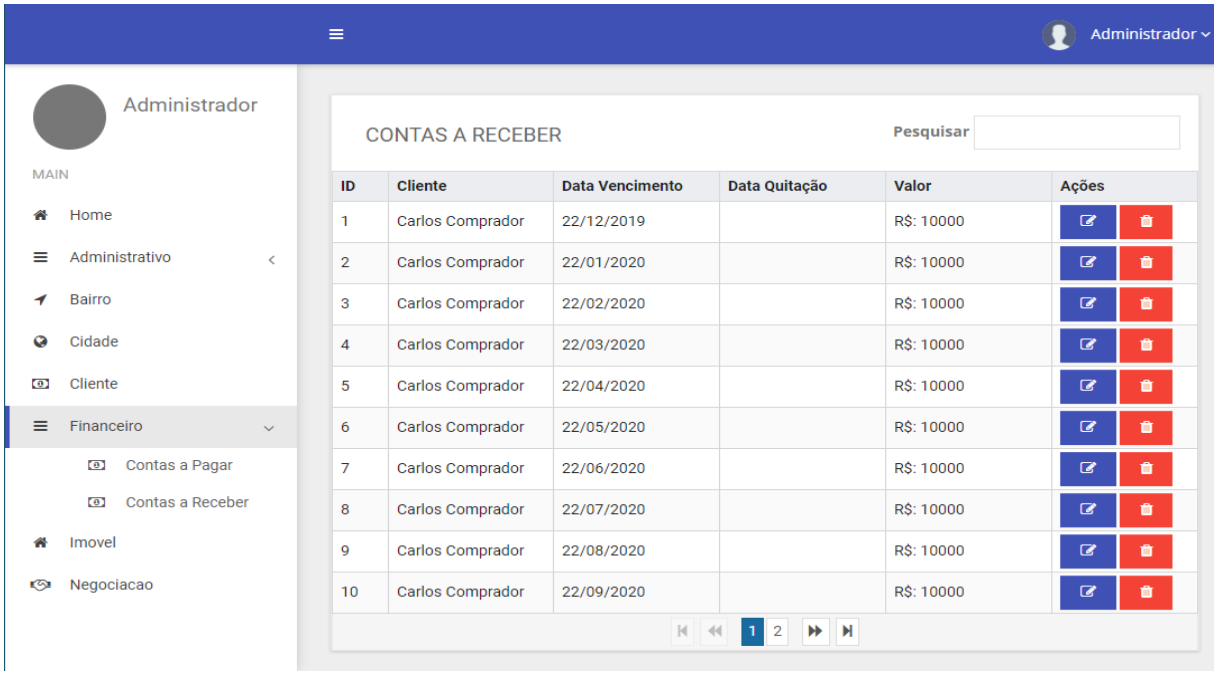
Figura 18 - Tela de confirmação de finalização de negociação



Fonte: Autoria própria.

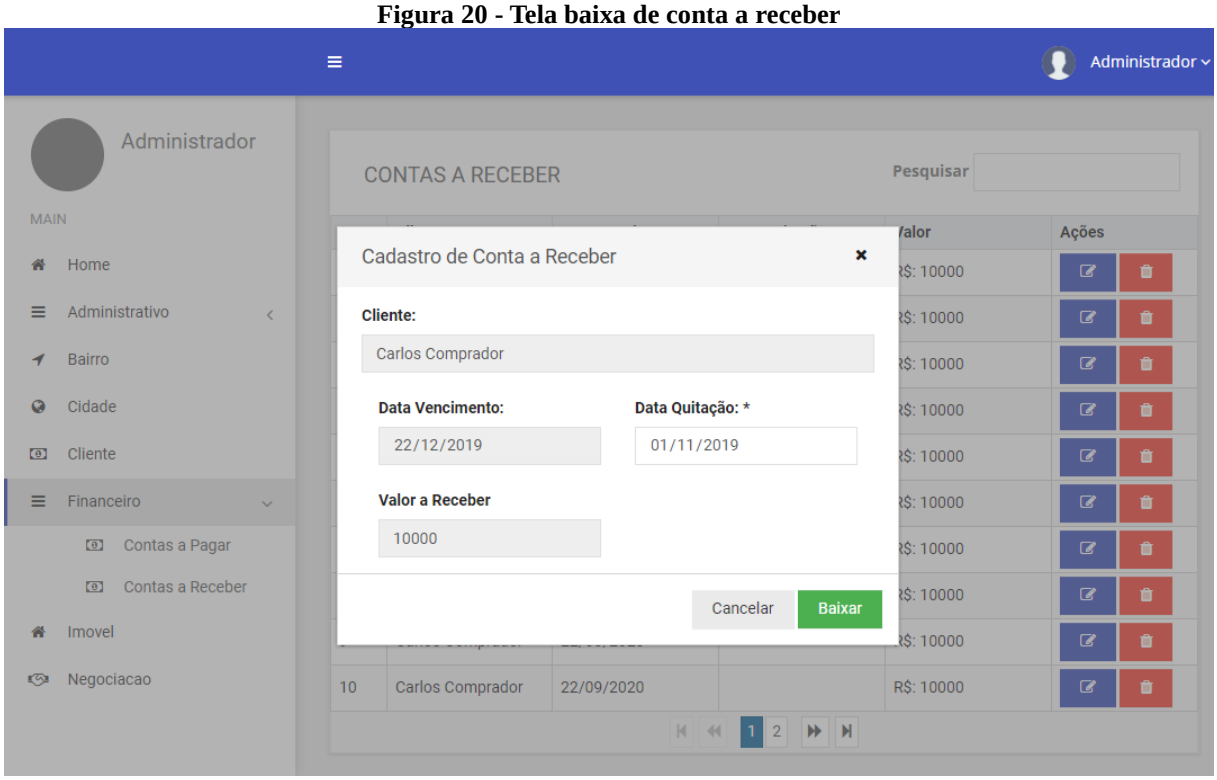
Assim que finalizada a negociação poderá ser observado a lista de parcelas na tela de contas a receber e/ou contas a pagar, conforme a Figura 19 e a Figura 20. Nessas telas não é possível incluir uma nova parcela manualmente, somente é possível quitar a parcela, ou excluí-la.

Figura 19 - Tela de contas a receber



Fonte: Autoria própria.

Figura 20 - Tela baixa de conta a receber



Fonte: Autoria própria.

As Figura 21 e Figura 22 apresentam as telas de acesso ao site de divulgação de imóveis, nelas é possível visualizar os imóveis disponíveis para venda ou aluguel, além disso

os usuários podem filtrar os registros por venda ou aluguel, quantidade de quartos e/ou vagas de garagem, cidade e bairro, e até mesmo valor.

Figura 21 - Tela do site

Fonte: Autoria própria.

Figura 22 - Tela do site

Fonte: Autoria própria.

4.4 DESENVOLVIMENTO DO SISTEMA

Nesta sessão serão apresentados os principais códigos para exemplificar o desenvolvimento do sistema deste trabalho, em especial, a utilização das tecnologias Spring Boot e Angular.

4.4.1 Aplicação *Back-end* Java utilizando Spring

Para o funcionamento do *Spring-Boot*, é necessário que seja criado uma classe com o método *main()* que fica no *package* br.edu.utfpr.pb.tcc2. Além de ser a classe de responsável por iniciar a aplicação, essa classe serve como configuração para informar para o Spring quais são os pacotes que deverão ser percorridos para encontrar os objetos, para que ocorra a injeção de dependências. Todos os outros pacotes devem ficar abaixo dessa classe. A Listagem 1 apresenta a classe Tcc2Aplication.java.

Listagem 1 - Classe Tcc2Aplication

```
package br.edu.utfpr.pb.tcc2;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
@SpringBootApplication
@Configuration
@EnableAutoConfiguration
@ComponentScan
public class Tcc2Application {
    public static void main(String[] args) {
        SpringApplication.run(Tcc2Application.class, args);
    }
}
```

Fonte: Autoria própria.

Na Figura 23 é possível visualizar a estrutura do projeto com o desenvolvimento do lado servidor da aplicação. Os principais pacotes da aplicação são:

Config: Arquivos de configurações do servidor, configurações de segurança e autenticação, além de formatadores de dados.

Controller: Classes responsáveis pelo tratamento das requisições HTTP enviadas pelo *front-end*.

Data: Repositórios de persistência.

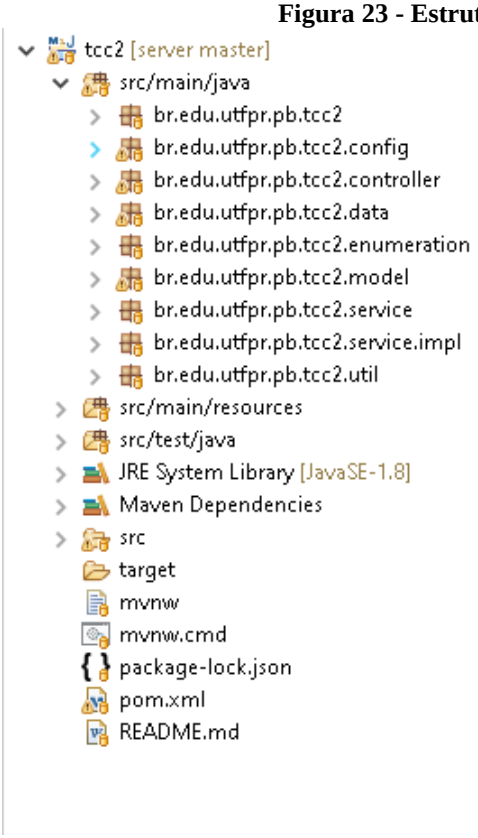
Enumeration: Classes de criação do tipo de dado abstrato Enum e permissões do sistema.

Model: Classes Java com o mapeamento objeto-relacional.

Service e Sevice.Impl: Classes responsáveis pela persistência dos dados.

Util: Classes responsáveis pela conversão de tipos de dados.

Figura 23 - Estrutura do projeto *Back-end* na IDE Eclipse



Fonte: Autoria própria.

Para a construção do projeto foi utilizado uma ferramenta desenvolvida pela Apache, conhecida como Maven. O Maven é responsável pelo gerenciamento das configurações e importação das dependências necessárias. A Listagem 2 apresenta parte do arquivo pom.xml, onde dependências são configuradas.

Listagem 2 - Arquivo pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>br.edu.utfpr.pb</groupId>
    <artifactId>tcc2</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>jar</packaging>

    <name>tcc2</name>
    <description>Demo project for Spring Boot</description>
```

```
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.0.0.RELEASE</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <project.reporting.outputEncoding>UTF-
8</project.reporting.outputEncoding>
        <java.version>1.8</java.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-cache</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jpa</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-rest</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-security</artifactId>
        </dependency>
    </dependencies>
```

Fonte: Autoria própria.

Na Listagem 3 é apresentado o arquivo application.properties que é responsável por armazenar as informações de configuração da aplicação, como por exemplo, os dados de conexão com o banco de dados. Além disso, no banco de dados foi criado um usuário com acesso restrito a somente a base de dados da aplicação desenvolvida.

Listagem 3 - Arquivo application.properties

```
server.port=8080

spring.datasource.url= jdbc:postgresql://localhost:5432/tcc2
spring.datasource.username=tcc2
spring.datasource.password=tcc2utfpr
spring.jpa.hibernate.ddl-auto=update
spring.cloud.refresh.refreshable=none
```

Fonte: Autoria própria.

Para o controle de segurança da aplicação foi utilizado o framework Spring Security. Na Listagem 4 é exibido o código que define as regras de acesso dos usuários às páginas do sistema. O sistema *web* possui os menus de todas as telas de acesso ao sistema, com as liberações e as restrições específicas de cada usuário. Por meio da do método antMatchers são definidas as páginas permitidas a cada usuário de acordo com seu tipo de permissão.

Listagem 4 – Arquivo ResourceServerConfig

```
@Override
    public void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests()
.antMatchers("/usuario/**").hasAnyRole("ADMIN,USUARIO_CRIAR,USUARIO_EDITAR,USUARIO_REMOVER")
.antMatchers("/perfil/**").hasAnyRole("ADMIN,PERFIL_CRIAR,PERFIL_EDITAR,PERFIL_REMOVER")
.antMatchers("/funcionario/**").hasAnyRole("ADMIN,FUNCIONARIO_CRIAR,FUNCIONARIO_EDITAR,FUNCIONARIO_REMOVER")
.antMatchers("/bairro/**").hasAnyRole("ADMIN,BAIRRO_CRIAR,BAIRRO_EDITAR,BAIRRO_REMOVER")
.antMatchers("/cidade/**").hasAnyRole("ADMIN,CIDADE_CRIAR,CIDADE_EDITAR,CIDADE_REMOVER")
.antMatchers("/cliente/**").hasAnyRole("ADMIN,CLIENTE_CRIAR,CLIENTE_EDITAR,CLIENTE_REMOVER")
.antMatchers("/contasapagar/**").hasAnyRole("ADMIN,CONTASAPAGAR_EDITAR,CONTASAPAGAR_REMOVER")
.antMatchers("/contasareceber/**").hasAnyRole("ADMIN,CONTASARECEBER_EDITAR,CONTASARECEBER_REMOVER")
.antMatchers("/imovel/**").hasAnyRole("ADMIN,IMOVEL_CRIAR,IMOVEL_EDITAR,IMOVEL_REMOVER")
.antMatchers("/negociacao/**").hasAnyRole("ADMIN,NEGOCIACAO_CRIAR,NEGOCIACAO_EDITAR,NEGOCIACAO_REMOVER")
                .anyRequest().authenticated();
}
```

Fonte: Autoria própria.

Para validar as requisições recebidas pelo servidor é criado uma classe responsável pela verificação e criação de token, ela é mostrada na Listagem 5.

Listagem 5 - Classe de configuração de autenticação

```
@Configuration
```

```
@EnableAuthorizationServer
public class AuthorizationServerConfig extends
AuthorizationServerConfigurerAdapter {

    @Autowired @Qualifier("authenticationManagerBean")
    private AuthenticationManager authenticationManager;

    @Autowired @Qualifier("usuarioServiceImpl")
    private UserDetailsService userDetailsService;

    @Override
    public void configure(ClientDetailsServiceConfigurer clients) throws
Exception {
        clients.inMemory()
                .withClient("app")
                .secret(new BCryptPasswordEncoder().encode("appsecret"))
                .autoApprove(true)
                .authorizedGrantTypes("authorization_code", "refresh_token",
"password", "client_credentials")
                .scopes("openid");
    }

    @Override
    public void configure(AuthorizationServerEndpointsConfigurer endpoints)
throws Exception {
        endpoints
                .authenticationManager(authenticationManager)
                .userDetailsService(userDetailsService);
    }

    @Override
    public void configure(AuthorizationServerSecurityConfigurer oauthServer)
throws Exception {
        oauthServer
                .tokenKeyAccess("permitAll()")
                .checkTokenAccess("permitAll()")
                .passwordEncoder(new BCryptPasswordEncoder());
    }

}
```

Fonte: Autoria própria.

Na Listagem 6, é exibida a classe de implementação do *service* de negociação, em que foi criado o método chamado saveNegociacao. Esse método é utilizado para finalizar uma negociação, e, de acordo com o tipo do negócio, irá chamar os métodos contaaPagarAluguel e contaaReceberAluguel, os quais geram as parcelas a pagar e a receber dos devidos clientes. Caso seja uma venda, somente é chamado o método contaaReceberVenda, que calcula o valor de comissão a receber e cria um registro de imóvel, com algumas informações do antigo dono para o novo.

Listagem 6 - Classe NegociacaoServiceImpl

```
@Service
public class NegociacaoServiceImpl extends CrudServiceImpl<Negociacao, Long>
implements NegociacaoService {

	@Autowired
	NegociacaoData negociacaoData;
	@Autowired
	private ContaaPagarService pagarService;
	@Autowired
	private ContaaReceberService receberService;
	@Autowired
	private ImovelService imovelService;
	@Override
	protected JpaRepository<Negociacao, Long> getData() {
		return negociacaoData;
	}
	@Override
	public Negociacao saveNegociacao(Negociacao negociacaoaux) {

		Negociacao negociacao = new Negociacao();

		negociacao = save(negociacaoaux);

		if (negociacao.getStatus() != StatusNegociacaoEnum.FINALIZADA) {
			negociacao.setDatafim(LocalDate.now());
			negociacao.setStatus(StatusNegociacaoEnum.FINALIZADA);
			negociacao.getImovel().setAtivo(Boolean.FALSE);
			negociacao.getImovel().setVenda(Boolean.FALSE);
			negociacao.getImovel().setLocacao(Boolean.FALSE);
			save(negociacao);

			if (negociacao.getTiponegocio()==TipoNegociacaoEnum.ALUGUEL) {
				pagarService.contaaPagarAluguel(negociacao);
				receberService.contaaReceberAluguel(negociacao);
			} else {
			receberService.contaaReceberVenda(negociacao);

				Imovel imovel = new Imovel();
				imovel.setCliente(negociacao.getCliente());
				imovel.setAtivo(Boolean.TRUE);
				imovel.setVenda(Boolean.FALSE);
				imovel.setLocacao(Boolean.FALSE);
				imovel.setVvenda(null);
				imovel.setVlocacao(null);
				imovel.setRua(negociacao.getImovel().getRua());
				imovel.setLatitude(negociacao.getImovel().
						getLatitude());
				imovel.setLongitude(negociacao.getImovel().
						getLongitude());
				imovel.setBairro(negociacao.getImovel().getBairro());
				imovel.setQuartos(negociacao.getImovel().getQuartos());
				imovel.setVgaragem(negociacao.getImovel().
						getVgaragem());
				imovel.setMterreno(negociacao.getImovel().
						getMterreno());
				imovel.setMimovel(negociacao.getImovel().getMimovel());
				imovel.setCep(negociacao.getImovel().getCep());
				imovel.setViptu(negociacao.getImovel().getViptu());
				imovel.setVcondominio(negociacao.getImovel().
```

```
                                        getVcondominio());
                                imovel.setVtaxas(negociacao.getImovel().getVtaxas());
                                imovel.setImagem(negociacao.getImovel().getImagem());
                                imovelService.save(imovel);
                        }
                }
                return negociacao;
        }
}
```

Fonte: Autoria própria.

A geração das contas a receber é feita de acordo com o tipo do negócio (venda ou aluguel). A Listagem 7 apresenta a codificação dos métodos. A geração das contas a pagar usa a mesma lógica de programação do método contaaReceberAluguel, porém criando registros na entidade ContaaPagar.

Listagem 7 - Métodos criação de contas a receber

```
@Service
public class ContaaReceberServiceImpl extends CrudServiceImpl<ContaaReceber,
Long> implements ContaaReceberService {
        @Autowired
        private ContaaReceberData contaaReceberData;
        @Override
        protected JpaRepository<ContaaReceber, Long> getData() {
                return contaaReceberData;
        }
        @Override
        public void contaaReceberAluguel(Negociacao negociacao) {

                for (int i = 0; i < 12; i++) {
                        ContaaReceber entity = new ContaaReceber();

                        LocalDate dataVenc = negociacao.getDatavenc();
                        dataVenc = dataVenc.plusMonths(i);
                        entity.setNegociacao(negociacao);
                        entity.setValorrec(negociacao.getVreceber());
                        entity.setStatus(StatusContaEnum.ARECEBER);
                        entity.setDatavenc(dataVenc);
                        save(entity);
                }
        }

        @Override
        public void contaaReceberVenda(Negociacao negociacao) {
                ContaaReceber entity = new ContaaReceber();
                entity.setNegociacao(negociacao);
                 entity.setValorrec((negociacao.getVreceber() –
                                negociacao.getVpagar()));
                 entity.setStatus(StatusContaEnum.ARECEBER);
                 entity.setDatavenc(negociacao.getDatavenc());
                 save(entity);
        }
          public List<ContaaReceber>
findByStatusAndNegociacaoImovelId(StatusContaEnum status, Long id) {
```

```
            return contaaReceberData.findByStatusAndNegociacaoImovelId(status,
id);
        }
}
```

Fonte: Autoria própria.

4.4.2 Aplicação *Front-end* utilizando Angular

A Figura 24 apresenta a estrutura dos arquivos na IDE do Visual Studio Code. O desenvolvimento do lado cliente partiu da estrutura básica do Angular Cli.

Dentre os arquivos da estrutura de arquivos do angular vale ressaltar alguns:

node_modules: Diretório responsável por armazenar as bibliotecas, quando é adcionado alguma configuração no arquivo *package.json*, ele gerencia os pacotes e as suas versões dentro desse diretório.

src: Diretório da aplicação, dentro dele é possivel criar os services, components, modules entre outros.

src/app:

app.component.css: Arquivo responsável pelo estilo do nosso módulo.

app.component.html: Arquivo HTML do nosso componente principal.

app.component.ts: Este arquivo dentro da arquitetura MVC, tem as mesmas responsabilidades das classes controllers.

app.module.ts: Arquivo de module default.

Environments: Diretório que possui dois arquivos **.**ts, um utilizado no ambiente de produção e outro para ambiente de desenvolvimento.

index.html: Esse é o arquivo root, dentro dele é rodado a SPA.

main.ts: Esse é o arquivo principal da nossa *Solution*, ele pode ser considerado o "bootstrap" da aplicação.

polyfills.ts: Esse arquivo funciona como um interpretador de código entre aplicação e navegadores web.

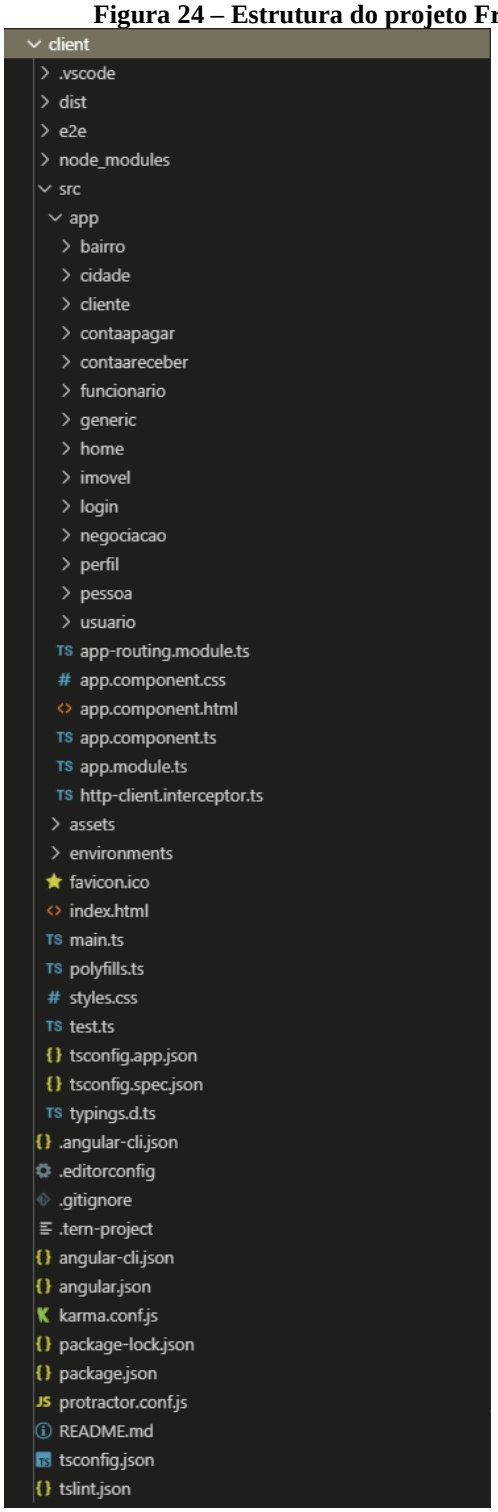
styles.css: Como todos os nossos componentes tem o seu próprio arquivo .css ou .scss, nós podemos utilizar esse arquivo para criar algo global como variáveis para nossa aplicação.

tsconfig.app.json e ***tsconfig.spec.json:*** arquivos de configuração de *TypeScript*.

typings.d.ts: Nesse arquivo nós podemos definir tipos para o *Typescript*.

tsconfig.json: Arquivo de configuração do *TypeScript*.

Figura 24 – Estrutura do projeto FrontEnd na IDE Visual Studio Code



Fonte: Autoria própria.

Como as telas possuem em maior parte as mesmas funções, foram implementadas classes abstratas que possuem todos os métodos necessários para ser utilizado nos componentes. A Listagem 8 apresenta a classe abstrata *CrudService*, que possui todos os

métodos responsáveis pelas requisições HTTP utilizadas na busca e envio de informações ao servidor.

Listagem 8 - CrudService

```
import {HttpClient} from '@angular/common/http';
import {Observable} from 'rxjs/Observable';
import {Page} from './page';

export abstract class CrudService<T, ID> {

  constructor(protected url: string, protected http: HttpClient) {
  }

  protected getUrl(): string {
    return this.url;
  }

  findAll(): Observable<T[]> {
    const url = `${this.getUrl()}`;
    return this.http.get<T[]>(url);
  }

  findPageable(page: number, size: number, order?: string, asc?: boolean): O
bservable<Page<T>> {
    let url = `${this.getUrl()}/page?page=${page}&size=${size}`;
    if (order) {
      url += `&order=${order}`;
    }
    if (asc !== undefined) {
      url += `&asc=${asc}`;
    }
    return this.http.get<Page<T>>(url);
  }

  findOne(id: ID): Observable<T> {
    const url = `${this.getUrl()}/${id}`;
    return this.http.get<T>(url);
  }

  save(t: T): Observable<T> {
    const url = `${this.getUrl()}`;
    return this.http.post<T>(url, t);
  }

  exists(id: ID): Observable<boolean> {
    const url = `${this.getUrl()}/exists/${id}`;
    return this.http.get<boolean>(url);
```

```
  }

  count(): Observable<number> {
    const url = `${this.getUrl()}/count`;
    return this.http.get<number>(url);
  }

  delete(id: ID): Observable<void> {
    const url = `${this.getUrl()}/${id}`;
    return this.http.delete<void>(url);
  }
}
```

Fonte: Autoria própria.

Na Listagem 9 pode ser visualizado o exemplo da classe *service* de um componente, todos os componentes estendem a classe CrudService. Nas classes *services* dos componentes é possível criar métodos personalizados, na classe utilizada como exemplo a BairroService, foi criado um método chamado findByCidade, em que é passado por parâmetro uma cidade para buscar somente os bairros vinculados àquela cidade.

Listagem 9 - Classe Service

```
import { Injectable } from '@angular/core';
import { CrudService } from '../generic/crud.service';
import { Bairro } from './bairro';
import { Cidade } from '../cidade/cidade';
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs/Observable';
import { environment } from '../../environments/environment';

@Injectable()
export class BairroService extends CrudService<Bairro, number> {

  constructor(http: HttpClient) {
    super(environment.api + '/bairro', http);
  }

  findByCidade(cidade: Cidade): Observable<Bairro[]> {
    const url = `${this.getUrl()}/findByCidade?id=${cidade.id}`;
    return this.http.get<Bairro[]>(url);
  }
}
```

Fonte: Autoria própria.

A seguir serão apresentados os códigos fontes associados ao *front-end* da aplicação. Na Listagem 10 é apresentado uma tabela gerada utilizando o componente p-*table* da biblioteca *PrimeNG*. O componente é responsável por gerar uma tabela com os dados retornados em formato JSON do *back-end*. Também pode-se observar a diretiva **ngIf*, que é utilizada para verificar se um elemento HTML deve ou não ser exibido, baseado em uma condição, neste caso as permissões anteriormente criadas e vinculadas à cada usuário do sistema.

Listagem 10 – Template de listagens de registros

```
 <p-table [value]="bairros" [paginator]="true" [rows]="10">
        <ng-template pTemplate="header">
          <tr>
            <th class="ids">ID</th>
            <th>Nome</th>
            <th>Cidade</th>
            <th class="acoes">Ações</th>
          </tr>
        </ng-template>
        <ng-template pTemplate="body" let-bairro>
          <tr>
            <td>{{bairro.id}}</td>
            <td>{{bairro.nome}}</td>
            <td>{{bairro.cidade?.nome}}</td>
            <td>
              <button *ngIf="hasRole('BAIRRO_EDITAR') || hasRole('ADMIN')" c
lass="btn btn-primary"
                (click)="editar(bairro)">
                <i class="fa fa-edit"></i>
              </button>
              <button *ngIf="hasRole('BAIRRO_REMOVER') || hasRole('ADMIN')"
class="btn btn-danger"
                (click)="confirmDelete(bairro)">
                <i class="fa fa-trash"></i>
              </button>
            </td>
          </tr>
        </ng-template>
      </p-table>
```

Fonte: Autoria própria.

Na Listagem 11 é apresentado um formulário de inclusão e edição de um cadastro. Nesse formulário a diretiva [(*ngModel*)] é utilizada para transportar valor de propriedades e retorno de métodos do componente para o template HTML. Essa propriedade do *framework*

Angular é chamada de *two-way data binding*, pois permite as atualizações dos valores entre o componente escrito em Typescipt e o o *template* escrito em HTML.

Listagem 11 - Formulário de inclusão e edição

```
 <p-
dialog  header="Cadastro de Bairros" [(visible)]="showDialog" mo dal="modal"
 [responsive]="true" [width]="500">

        <div class="row">
          <div class="col-sm-12">
            <div class="form-group">
              <label for="lblnome">Nome: *</label>
              <input id="nome" name="nome" class="form-
control" required minlength="4" [(ngModel)]="bairroEdit.nome">


            </div>
          </div>
        </div>
        <div class="row">
          <div class="col-sm-12">
            <div class="form-group">
              <label for="cidade">Cidade: *</label>
              <p-
dropdown name="cidade" id="cidade" placeholder="Selecione uma cidade" [optio
ns]="cidades"
                [(ngModel)]="bairroEdit.cidade" optionLabel="nome" required
[autoDisplayFirst]="false">
              </p-dropdown>
            </div>
          </div>
        </div>

        <p-footer>
          <button type="button" class="btn btn-
default" (click)="cancelar()">Cancelar</button>
          <button type="button" class="btn btn-
success" (click)="salvar()">Salvar</button>
        </p-footer>
      </p-dialog>
```

Fonte: Autoria própria.

A Listagem 12 apresenta o codigo de implementação do serviço responsável pela autenticação e validação de *token*, para que os usuários consigam autenticar-se na aplicação.

Listagem 12 - Classe LoginService

```
import { Injectable } from '@angular/core';
import { HttpClient, HttpHeaders } from '@angular/common/http';
import { ActivatedRouteSnapshot, CanActivate, Router, RouterStateSnapshot } fr
om '@angular/router';
import { Observable } from 'rxjs/Observable';
import 'rxjs/add/operator/map';
import 'rxjs/add/operator/catch';
import { ErrorObservable } from 'rxjs/observable/ErrorObservable';
import { Subject } from 'rxjs/Subject';
import { AccessToken } from './access.token';
import { environment } from '../../environments/environment';

@Injectable()
export class LoginService implements CanActivate {

  userInfo: any;
  isAuthenticated = new Subject<boolean>();

  constructor(private http: HttpClient, private router: Router) {
  }

  canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): Obse
rvable<boolean> | Promise<boolean> | boolean {
    const url = `${environment.api}/user-info`;
    return this.http.get(url).map(e => {
      this.userInfo = e;
      this.isAuthenticated.next(true);
      return true;
    }).catch(() => {
      this.loggout();
      return new ErrorObservable('User is not authenticated');
    });
  }

  getUserInfo(): any {
    return this.userInfo;
  }

  hasRole(role: string): boolean {
    if (this.getUserInfo() && this.getUserInfo().authorities) {
      return this.getUserInfo().authorities.filter(e => e.authority === 'ROLE_
' + role).length > 0;
    }
    return false;

  }

  logout() {
    Object.keys(new AccessToken()).forEach(key => localStorage.removeItem(key)
```

```
);
    this.isAuthenticated.next(false);
    this.userInfo = null;
    this.router.navigate(['/login']);
  }

  login(username: string, password: string): Observable<AccessToken> {
    const params = new URLSearchParams();
    params.append('username', username);
    params.append('password', password);
    params.append('grant_type', 'password');
    const headers = new HttpHeaders({
      'Content-type': 'application/x-www-form-urlencoded',
      'Authorization': 'Basic ' + btoa(`app:appsecret`)
    });

    return this.http.post<AccessToken>(`${environment.api}/oauth/token`, param
s.toString(), { headers: headers })
      .map(e => {
        Object.keys(e).forEach(key => localStorage.setItem(key, e[key]));
        this.isAuthenticated.next(true);
        return e;
      });
  }
}
```

Fonte: Autoria própria.

A Listagem 13 apresenta a implementação do *inteceptor* que identifica todas as requisições do tipo HTTP e verifica se a mesma possui um *token* válido. Portanto, caso o usuário tenha clicado em sair do sistema ou o tempo de sessão tenha expirado o mesmo será direcionado a página de autenticação.

Listagem 13 – Classe interceptor

```
import { Injectable } from '@angular/core';
import 'rxjs/add/operator/do';
import { HttpEvent, HttpHandler, HttpInterceptor, HttpRequest } from '@angular
/common/http';
import { Observable } from 'rxjs/Observable';
import { Router } from '@angular/router';

@Injectable()
export class HttpClientInterceptor implements HttpInterceptor {

  constructor(private router: Router) {
  }
```

```
  intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<an
y>> {
    const token = localStorage.getItem('access_token');

    if (token && !req.headers.has('Authorization')) {
      const authReq = req.clone({
        headers: req.headers.set('Authorization', 'Bearer ' + token)
      });

      return next.handle(authReq)
        .do(() => {
        }, err => {
          if (err.status === 401) {
          }
        });
    }
    return next.handle(req);
  }
}
```

Fonte: Autoria própria.

Aplicações desenvolvidas em Angular utilizam o conceito de *Simple-Page Application* (SPA), portanto foi necessário implementar um arquivo de configuração de rotas como pode ser visualizado na Listagem 14. Para que seja possível que o usuário navegue entre as páginas do sistema foi definida a *Uniform Resource Locator* (URL) das páginas e qual componente será responsável pela mesma.

Listagem 14 - Classe configurações de rotas

```
import {UsuarioComponent} from './usuario/usuario.component';
import {RouterModule, Routes} from '@angular/router';
import {ModuleWithProviders} from '@angular/core';
import {LoginService} from './login/login.service';
import {LoginComponent} from './login/login.component';
import {HomeComponent} from './home/home.component';
import {ClienteComponent} from './cliente/cliente.component';
import {PerfilComponent} from './perfil/perfil.component';
import {FuncionarioComponent} from './funcionario/funcionario.component';
import {BairroComponent} from './bairro/bairro.component';
import {CidadeComponent} from './cidade/cidade.component';
import {ImovelComponent} from './imovel/imovel.component';
import {NegociacaoComponent} from './negociacao/negociacao.component';
import {ContaaReceberComponent} from './contaareceber/contaareceber.component';
import {ContaaPagarComponent} from './contaapagar/contaapagar.component';
const routes: Routes = [
  {
```

```
    path: '', canActivate: [LoginService], children: [
      {path: '', component: HomeComponent},
      {path: 'perfil', component: PerfilComponent},
      {path: 'usuario', component: UsuarioComponent},
      {path: 'cliente', component: ClienteComponent},
      {path: 'funcionario', component: FuncionarioComponent},
      {path: 'bairro', component: BairroComponent},
      {path: 'cidade', component: CidadeComponent},
      {path: 'imovel', component: ImovelComponent},
      {path: 'negociacao', component: NegociacaoComponent},
      {path: 'contaareceber', component: ContaaReceberComponent},
      {path: 'contaapagar', component: ContaaPagarComponent}
    ]
  },
  {path: 'login', component: LoginComponent}
];

export const AppRouting: ModuleWithProviders = RouterModule.forRoot(routes);
```

Fonte: Autoria própria.

5 CONCLUSÃO

O desenvolvimento deste projeto teve como objetivo fornecer um software de gestão imobiliário agregando qualidade e agilidade nos processos administrativos. Percebendo o gerenciamento de imóveis, locações e venda de imóveis, além de vários outros cadastros adicionais. A necessidade em muitas empresas de pequeno e médio porte em possuir um software de gestão que facilite a união das informações ora espalhadas, em um só lugar, assim gerando economia de tempo e reduzindo retrabalho.

O desenvolvimento web permite aos usuários acesso ao sistema de qualquer máquina com acesso à internet, seja o acesso via computador, celular ou *tablet*, dessa forma simplificando o acesso.

O desenvolvimento deste trabalho possibilitou grande aprendizado, relacionado tanto a modelagem, análise de requisitos, quanto a implementação do sistema. Durante todo esse processo, foram realizados vários testes e alterações até chegar a um resultado mais próximo possível do esperado.

Durante o desenvolvimento do trabalho foram encontradas diversas dificuldades, entre as que se destacam posso citar a geração das parcelas a pagar e receber na finalização das negociações, além do mapa e inclusão de arquivos no cadastro do imóvel. Grande parte das dificuldades foi devido ao tempo de aprendizagem de utilização do *Framework* Angular.

Para dar sequência a esta aplicação, futuramente podem ser implementadas novas funcionalidades para que a utilização desta possa trazer maiores benefícios e informações aos usuários, tais como, a geração dos relatórios, melhorias nos cadastros de acordo com regras de negócio por meio de uma análise de requisitos mais aprofundada e o site para prospecção de novos clientes.

REFERÊNCIAS

AGUSTIN, José Luis Herrero. Model-driven web applications. **Science and Information Conference**, 2015, p. 954-964.

BAZOTTI, Cristiane; GARCIA, Elias. **A importância do sistema de informação gerencial na gestão empresarial para tomada de decisões.** Disponível em: http://saber.unioeste.br/index.php/csaemrevista/article/view/368/279 Acesso em: 28 ago. 2018.

BERNARDI, Mario Luca; DI LUCCA, Giuseppe Antonio; DISTANTE, Damiano. **Model-driven fast prototyping of RIAs: from conceptual models to running applications.** Disponível em: https://ieeexplore.ieee.org/document/6968522. Acesso em: 17 set. 2018.

CASTELEYN, Sven; GARRIGÓS, Irene; MAZÓN, Jose-Norberto**. Ten years of rich internet applications: a systematic mapping study, and beyond.** ACM Transactions on the Web, v. 8, n. 3, art 18, 2014. Disponível em: https://core.ac.uk/download/pdf/32324020.pdf. Acesso em: 17 set. 2018.

FIELDING, Roy Thomas. **Architectural Styles and the Design of Network-based Software Architectures**, 2000. Disponível em: https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm. Acesso em: 21 set. 2018.

JEREMY, Allaire. **Macromedia flash MX - A next-generation rich client. Macromedia White Pape**r. March 2002. Disponível em: http://download.macromedia.com/pub/flash/whitepapers/richclient.pdf. Acesso em: 17 set. 2018.

MELIÁ, Santiago, GÓMEZ, Jaime; PÉREZ, Sandy; DÍAZ, Oscar. **Facing Architectural and Technological Variability of Rich Internet Applications**. IEEE Internet Computing, 2010, p. 1-7. Disponível em: https://ieeexplore.ieee.org/document/5481364/ Acesso em: 17 set. 2018.

SAUDATE, Alexandre. **REST: Construa API's inteligentes de maneira simples.** Casa do código, 2013.

SECOVI, Departamento de Economia e Estatística do Secovi-SP (Sindicato da Habitação). **Mercado imobiliário de São Paulo mantém bons resultados em junho.** Disponível em: http://www.secovi.com.br/downloads/pesquisas-e-indices/pmi/2018/arquivos/201806-pmi.pdf. Acesso em: 28 ago. 2018