INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DO RIO GRANDE DO SUL CAMPUS CANOAS
CURSO SUPERIOR DE TECNOLOGIA EM ANÁLISE E DESENVOLVIMENTO DE SISTEMAS

ANDERSON FUHR SOUZA

Lar Certo Imóveis: Aplicação web para gestão de imóveis

Canoas 2024

ANDERSON FUHR SOUZA

Lar Certo Imóveis: Aplicação web para gestão de imóveis

Trabalho de Conclusão de Curso apresentado como requisito parcial para obtenção do grau de Tecnólogo em Análise e Desenvolvimento de Sistemas pelo Instituto Federal de Educação,Ciência e Tecnologia do Rio Grande do Sul - Campus Canoas.

Orientador: Dra. Carla Odete Balestro Silva

Canoas 2024

SUMÁRIO

LISTA DE ILUSTRAÇÕES

LISTA DE ABREVIATURAS E SIGLAS

API	Application Programming Interface (Interface de Programação de Aplicações)
CI/CD	Continuous Integration and Continuous Deployment/Delivery (Integração Contínua e Entrega/Implantação Contínua)
DTO	Data Transfer Object (Objeto de Transferência de Dados)
ERP	Enterprise Resource Planning (Planejamento de Recursos Empresariais)
HTTP	Hypertext Transfer Protocol (Protocolo de Transferência de Hipertexto)
IBGE	Instituto Brasileiro de Geografia e Estatística
JPA	Java Persistence API
JSON	JavaScript Object Notation
JVM	Java Virtual Machines - Máquina Virtual Java
JWT	JSON Web Token
LGPD	Lei Geral de Proteção de Dados
MVC	Model View Controller(Modelo Visão Controlador)
MVP	Minimum Viable Product (Produto Mínimo Viável)
REST	Representational State Transfer
SQL	Structured Query Language - Linguagem de Consulta Estruturada
URL	Uniform Resource Locator (Localizador Uniforme de Recursos)
VCS	Version Control System - Sistema de Controle de Versionamento
WWW	World Wide Web

1 INTRODUÇÃO

Segundo Souza, J. P. de, e Figueiredo (2014), a implementação de um sistema de informação adequado é essencial para auxiliar no gerenciamento e na organização de dados no setor imobiliário. Um sistema voltado ao cadastro de clientes e imóveis promove a automatização dos processos empresariais, facilitando a tomada de decisões estratégicas e o controle operacional. Nas empresas de pequeno e médio porte, onde os processos frequentemente dependem de ferramentas manuais, os desafios tornam-se ainda mais evidentes, resultando em morosidade, falta de eficiência, maior risco de erros operacionais e aumento de custos.

Neste contexto, o presente trabalho propõe o desenvolvimento de uma ferramenta online de gestão de imóveis para a “Lar Certo Imóveis”, visando informatizar e otimizar as principais operações da imobiliária, atendendo às necessidades específicas da empresa.

Em análise técnica foi verificado que atualmente, a empresa utiliza planilhas e documentos físicos para gerenciar as operações, como controle de contratos de aluguéis, administração de despesas e manutenção de imóveis. Nesta perspectiva, visualizamos a necessidade de um sistema personalizado, para o gerenciamento dos dados, que integre as necessidades globais da empresa, respectivamente o seguimento financeiro, contábil, eixos estruturais de compra, venda e aluguel, visando à automação e à otimização dos processos organizacionais.

Para o detalhamento desse trabalho, será considerado como desafio à análise de requisitos, à escolha das tecnologias adequadas, à modelagem de dados, à segurança da informação e à integração entre sistemas e tecnologias.

Visualiza-se que o sistema desenvolvido proporcione benefícios significativos à empresa, como a redução de custos, a eliminação de redundâncias e retrabalhos, o aumento da produtividade e o suporte à tomada de decisões estratégicas.

Desta forma, o presente trabalho apresentará os resultados obtidos com a implementação do sistema na empresa “Lar Certo Imóveis”, bem como as contribuições e oportunidades de exploração desse segmento. Acredita-se que este estudo contribuirá para o conhecimento acadêmico, oferecendo *insights* valiosos para projetos futuros que busquem desenvolver ou aprimorar sistemas de gestão voltados ao setor imobiliário. Além disso, fornecerá uma análise aprofundada do desenvolvimento de um sistema projetado para aprimorar a eficiência operacional e a capacidade de adaptação às demandas de um mercado imobiliário em constante crescimento.

2 MOTIVAÇÃO

A TOTVS é uma empresa brasileira que, segundo Meirelles (2022), é uma das maiores empresas de tecnologia da América Latina. Detentora de diversos softwares, a empresa tem como principais soluções os sistemas ERP, como Protheus, RM, Datasul e Logix, conforme informações disponíveis no site oficial da empresa.

De acordo com Sumner (2005), os sistemas ERP têm como objetivo essencial integrar os processos empresariais por meio de um banco de dados único, promovendo eficiência ao evitar a duplicação de informações. Além disso, esses sistemas são modulares, o que permite adaptações às necessidades específicas de cada organização. Eles automatizam fluxos de trabalho, desde a entrada de pedidos até o pagamento de faturas, e fornecem relatórios e análises que embasam as decisões gerenciais.

Para Khaleel Hayes (HAYES, 2024), um módulo financeiro de ERP é um software que consolida dados financeiros e gera relatórios detalhados. Esse módulo permite que informações financeiras sejam comunicadas de forma clara para parceiros externos, como fornecedores e clientes, quando necessário. Uma das maiores vantagens dos sistemas ERP, segundo Chopra (2003), é sua estrutura modular: os módulos compartilham uma base de dados única, operam de forma independente e são customizáveis e escaláveis.

No entanto, embora os sistemas ERP sejam amplamente reconhecidos por sua capacidade de integrar processos e otimizar operações, a sua implementação em empresas de pequeno e médio porte apresenta desafios significativos, como custos elevados, complexidade e a presença de funcionalidades que excedem as necessidades reais. Essas limitações tornam as soluções tradicionais de mercado inviáveis para muitas dessas empresas.

Nesse contexto, considerando a importância das empresas de pequeno porte (EPP) para a promoção da sustentabilidade e o desenvolvimento local, além de seu potencial de crescimento e competitividade no mercado, este trabalho propõe a criação de uma aplicação web personalizada para a empresa “Lar Certo Imóveis”. Essa solução se apresenta como uma alternativa acessível e eficaz, voltada para atender às necessidades específicas da organização. Inspirado nos conceitos de modularidade e integração dos sistemas ERP, o sistema proposto busca centralizar e automatizar operações essenciais, proporcionando maior eficiência e organização. Ademais, o desenvolvimento interno da solução garante flexibilidade para atender demandas específicas da empresa, eliminando custos desnecessários e promovendo independência tecnológica.

3 PROBLEMA E PROPOSTA DE SOLUÇÃO

Existem empresas de pequeno porte, de âmbito familiar, com poucos funcionários ou até mesmo um indivíduo administrando a gestão dos negócios. Como exemplo, há a empresa Lar Certo Imóveis. A empresa é administrada por uma família, onde pai e filha administram juntos. Os dois têm como maior foco manter a gestão das dezenas de imóveis da família.

A Lar Certo Imóveis possui uma rede de apoio de colaboradores em diversos segmentos, tais como, imobiliárias, empreiteiros e contadores que auxiliam na gestão dos imóveis aos quais a empresa inclusive tem o mesmo imóvel compartilhado entre diversas imobiliárias, que concorrem entre si para arrendar o imóvel, o que torna muitas vezes o controle ainda mais difícil.

A principal responsabilidade da família está concentrada na fiscalização e na tomada de decisões relacionadas à gestão dos imóveis, como contabilizar os custos, manter a manutenção dos imóveis em dia, gerenciar o contato com inquilinos e diversas imobiliárias, além de criar um centro de custo e planejamento estratégico. No entanto, essas tarefas tornam-se muito onerosas e complexas devido à ausência de um sistema de gestão centralizado, capaz de atender às necessidades específicas da empresa.

Atualmente, a empresa organiza-se por meio de planilhas eletrônicas no Microsoft Excel e documentos físicos. No entanto, como as planilhas do Excel exigem um conhecimento mais avançado, um dos integrantes da família enfrenta dificuldades para manipulá-las, o que resulta em uma dependência significativa de outras empresas, como imobiliárias, para realizar o controle de contas a pagar. Além disso, a empresa não possui relatórios consolidados sobre custos e receitas, como gastos com manutenção, pagamento de dívidas dos imóveis e os valores gerados pelos aluguéis.

A empresa tem a necessidade de um sistema que seja personalizável, escalável e de fácil utilização. Nesse contexto, a implementação de um ERP surge como uma excelente solução, permitindo à empresa gerenciar seu negócio de maneira prática e eficiente, reduzindo a dependência de fontes externas de gestão e informação. Além disso, o ERP possibilitará a centralização dos dados, facilitando o acesso às informações e promovendo maior organização e controle sobre as operações da empresa.

É difícil fornecer uma estimativa precisa do valor mínimo para a licença de ERPs. O Protheus, da empresa TOTVS, por exemplo, pode variar de centenas a milhares de reais, uma vez que seus custos de licenciamento dependem de diversos fatores, como o número de usuários, o tipo de licença, os módulos ou funcionalidades específicas necessárias, entre outros (TOTVS, 2024a).

A modelagem de dados do Protheus, por exemplo, é definida pela TOTVS como parte do software padrão. No entanto, quando você compra uma licença do Protheus, você pode personalizar a modelagem de dados para atender às suas necessidades de negócios

específicas, mas isso tem um alto custo para pequenas empresas, que dificilmente têm capacidade orçamentária para realizar tais modificações. (TOTVSb, 2024).

Em resumo, softwares como o Protheus podem vir com uma gama variada de recursos que não serão utilizados pela empresa ou que terão alto custo de personalização, que acabam por tornar seu uso muito complexo ou inviável pelo alto custo.

A proposta de solução é, portanto, o desenvolvimento de uma aplicação WEB[1] que seja simples de usar, com um cadastro de imóveis contendo todas as suas características e a possibilidade de gerenciar os imóveis e controle do caixa da empresa.

O sistema deverá contar inicialmente com um sistema de login que deve controlar os diferentes perfis de usuários, que terão permissões de acessos diferenciados, isso irá garantir que no futuro seja possível o cadastro de locatários ou prestadores de serviços que terão diferentes permissões de acesso ao sistema, garantindo escalabilidade ao sistema.

Foi implementado gráficos que apresentam uma análise visual da demonstração de resultados do exercício (DRE), assim, a empresa pode controlar de forma eficiente o fluxo de caixa, identificando tendências e ajustando suas estratégias financeiras conforme necessário.

As telas de cadastro e consulta devem ser simples e apenas com validações e campos estritamente necessárias à aplicação.

O sistema deverá ter a possibilidade de ser escalável, desta forma deverá ser possível acoplar novas funcionalidades sem afetar as demais.

Auxiliar na gestão de contratos, sendo possível acompanhar e filtrar imóveis que estão locados e seus prazos de contratos pré-estabelecidos, para análise e projeção de metas e estratégias.

Será utilizado React no front-end e Java com o framework Spring Boot no back-end, com o banco de dados PostgreSQL e o controle de versionamento do banco de dados realizado por meio do Flyway. Dessa forma, será possível estruturar a aplicação de forma escalável, dividindo-a em componentes e microsserviços que se comunicarão utilizando o padrão REST(Representational State Transfer)[2]. Esses microsserviços poderão ser implementados em módulos, aproximando o sistema do conceito de um ERP.

---

1 Refere-se ao sistema de documentos e informações interligados por meio da Internet.

2 Transferência de Estado Representacional.

4 OBJETIVO

O presente capítulo abordará os objetivos compreendidos pela proposta, dividida em duas partes: objetivo geral e objetivos específicos.

4.1 OBJETIVO GERAL

O presente projeto busca desenvolver uma ferramenta on-line de gestão de imóveis para auxiliar a Lar Certo Imóveis.

4.2 OBJETIVO ESPECÍFICO

Compreender as necessidades e o entendimento dos colaboradores das empresas em relação à tecnologia empregada.

Investigar e selecionar tecnologias e arquiteturas adequadas para o desenvolvimento da aplicação, visando a construção de um MVP (Produto Mínimo Viável).

Utilizar ferramentas e automações para estabelecer uma pipeline de CI/CD (Continuous Integration and Continuous Deployment/Delivery)[1].

Implementar ferramentas de gerenciamento de versão de código e banco de dados, como Git e Flyway.

Aplicar modelos de padrões de projeto como SOLID, Clean Code e Design Patterns para garantir que o sistema seja escalável, manutenível e de alta qualidade.

Documentar o processo de implementação da ferramenta.

[1] Integração Contínua/Entrega Contínua

5 ESTADO DA ARTE

O presente capítulo visa estudar casos de uso de sistemas para gestão de imobiliárias.

5.1 TECIMOD

Tecimod é uma empresa brasileira criada em 2012 e atualmente sediada em Santa Catarina, que fornece um sistema web e aplicativo mobile que permite a gestão de imóveis, clientes, negociações, contratos, além de possuir módulos de gestão financeira e de equipe de vendas(TECIMOB, 2024).

Seu sistema modularizado permite ao cliente inserir algumas funcionalidades conforme sua necessidade, porém não há acesso ou suporte, por exemplo, ao código-fonte da aplicação, controle sobre hospedagem ou modelagem do banco de dados, impossibilitando a personalizações ou inserção pontual de mais ferramentas.

Alguns recursos do sistema são pagos como extra como a inclusão de cada usuário a mais que deve ser pago taxa extra, mas oferecem um site completo com cadastro de imóveis ilimitados, plugin[1] com o whatsapp, integração com diversos portais, ficha de visitas, marca d´água nas fotos cadastradas, anexo de documentos e a garantia de comprimento da LGPD(Lei Geral de Proteção de Dados).

Como visto na lista anterior que mostra alguns dos recursos disponíveis, é um sistema e aplicativo imobiliário prático, porém focado na venda e comercialização dos imóveis via website.

A empresa foca na usabilidade e design, podendo personalizar a logomarca e designe do site escolhendo alguns modelos disponíveis.

Porém, os sites produzidos seguem o template de scrolling website[2], padrão amplamente utilizado na internet, como exemplo pode ser citado os sites de das empresas Apple[3] e/ou Tesla[4].

Podemos observar nas Figuras 1 e 2 exemplos de sites divulgados no portfólio da Tecimob, que apresentam um design personalizado no cabeçalho, com algumas estilizações específicas. No entanto, nas Figuras 3 e 4, é possível identificar o reaproveitamento do mesmo componente de amostra de vídeos em ambos os sites, evidenciando o uso da componentização (TECIMOB, 2024).

---

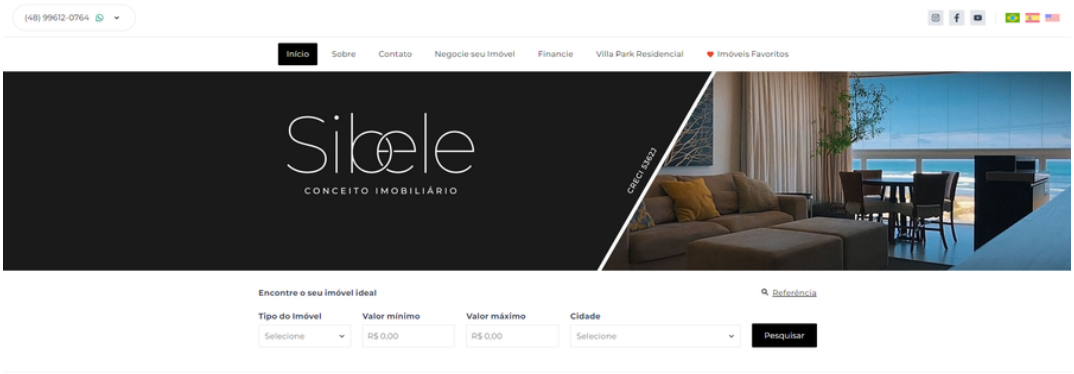
1 Componente de software que adiciona funcionalidades extras a um programa principal.

2 Termo que se refere a um tipo de website em que o conteúdo é apresentado em uma única página que pode ser percorrida através da rolagem do mouse ou do toque em dispositivos móveis, em vez de clicar em links para acessar outras páginas.

3 Disponível em: https://www.apple.com/. Acessado em 09 jun. 2024.

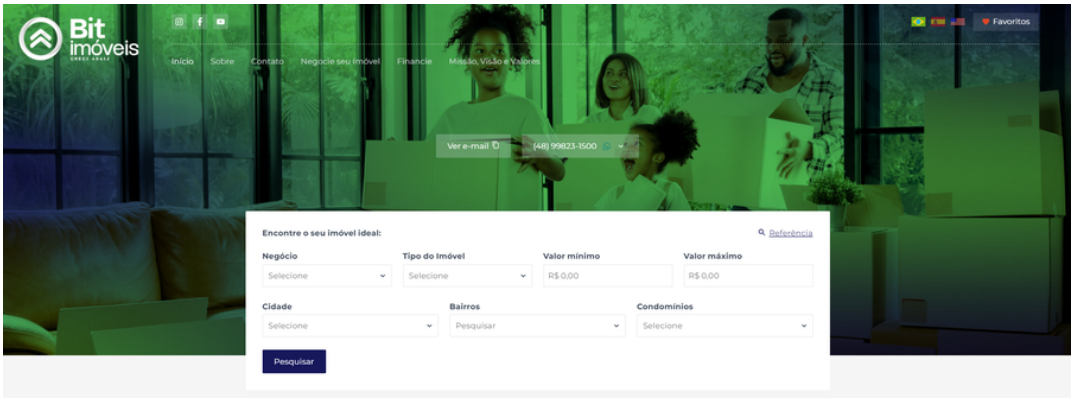
4 Acesso: https://www.tesla.com/. Acessado em 09 jun. 2024.

Figura 1 – Cabeçalho do site https://sibeleimoveis.com.br/



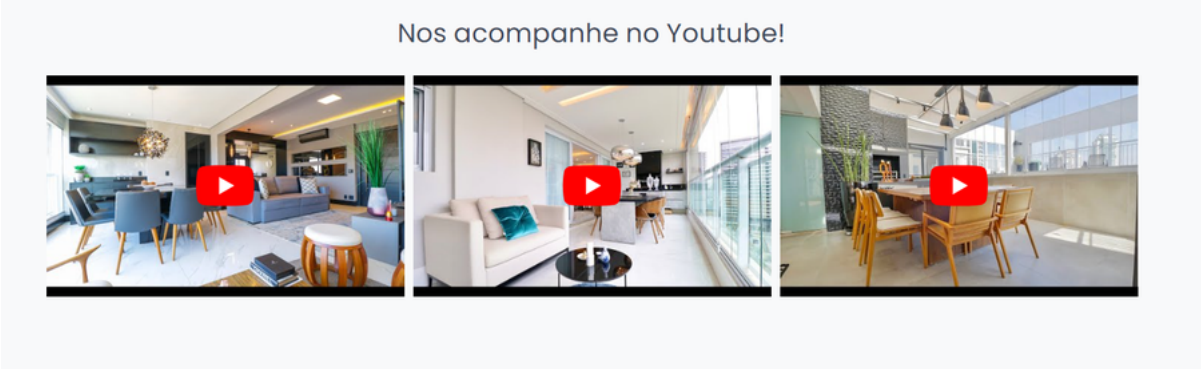
Fonte: https://sibeleimoveis.com.br/

Figura 2 – Cabeçalho do site https://bitimoveis.com/



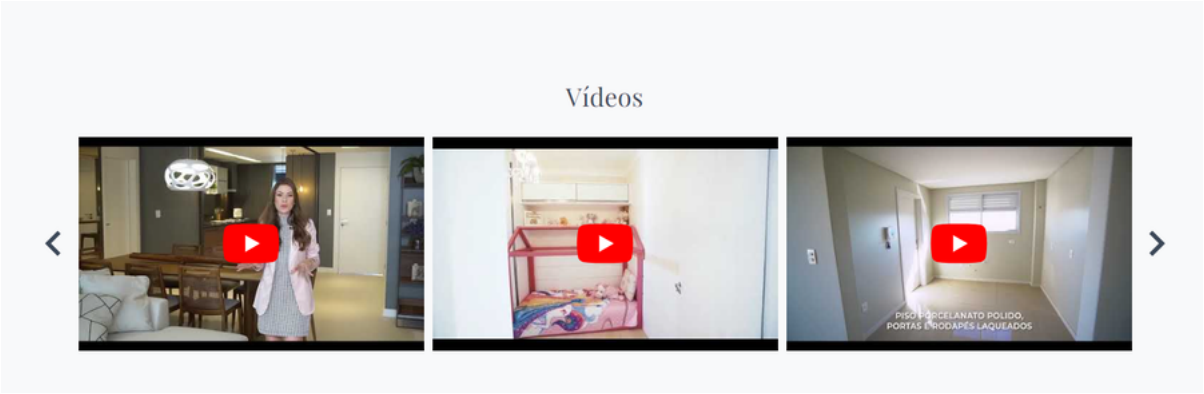
Fonte: https://bitimoveis.com/

Figura 3 – Seção de vídeos do site: https://sibeleimoveis.com.br



Fonte: https://sibeleimoveis.com.br

Figura 4 – Seção de vídeos do site: https://bitimoveis.com



Fonte: https://bitimoveis.com

5.2 IMOBILIÁRIA21

A Group Software atua em 15 países e mais de 250 cidades no Brasil, com escritórios em diversas localidades, incluindo Belo Horizonte, São Paulo e Nova York. Ela é proprietária do software de gestão imobiliária Imobiliária21, especializado na administração de locações de imóveis, oferecendo funcionalidades para controle financeiro, gestão de contratos e inadimplência, e dashboards intuitivos que centralizam informações essenciais.(GROUP S., 2024)

O Imobiliária21 permite o cadastro detalhado de pessoas e imóveis, o controle de contratos e encargos, além da automatização de processos como reajustes de aluguéis e vencimentos. As funcionalidades também incluem faturamento, cobrança, emissão de boletos, controle de inadimplência e geração de relatórios financeiros. Embora ofereça uma gestão abrangente, o software não possui suporte para estilização ou integração com websites de anúncios de imóveis, necessitando de pacotes adicionais para esse fim.(IMOBILIARIA 21, 2024)

5.3 IMOBZI

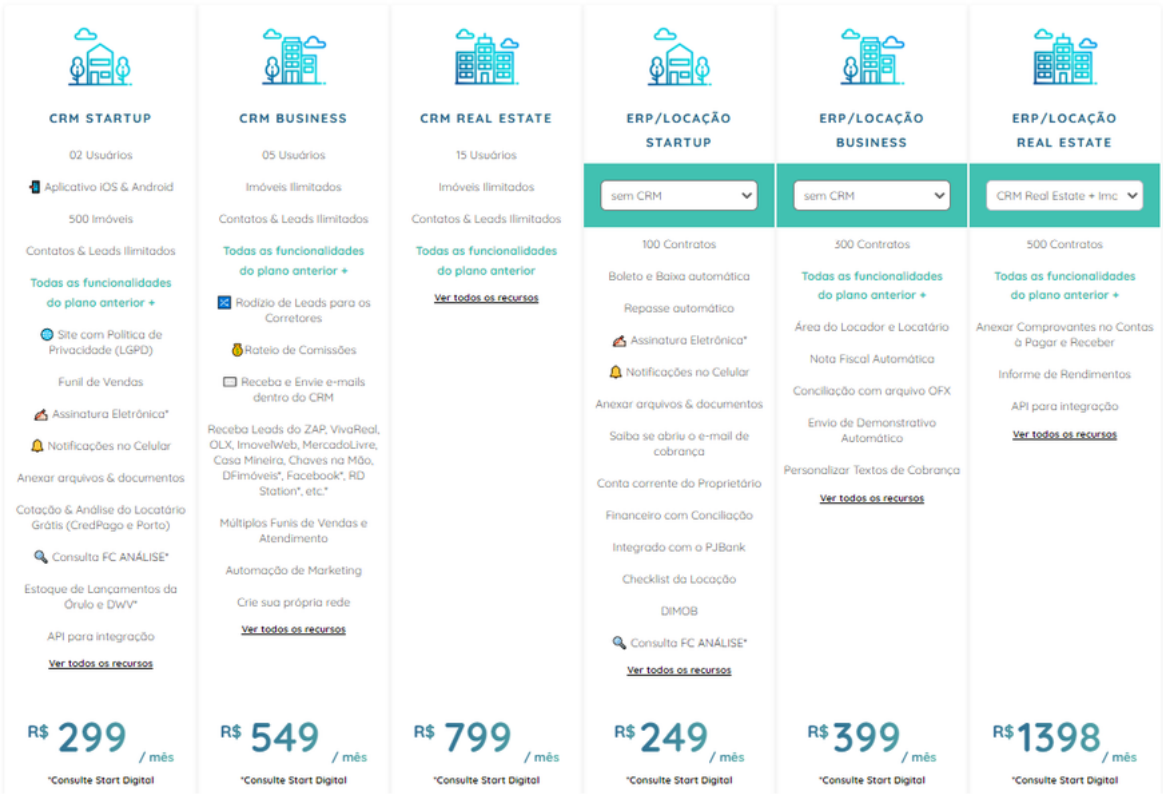
A Imobzi, é uma plataforma completa para digitalização de Imobiliárias e corretores de imóveis. Criada em 2004 é uma plataforma que reúne em um só lugar, aplicativo para ERP com Gestão de Locação e Venda, Financeiro, Site para Imobiliárias e Esteira Digital para Locações (IMOBZI, 2024a).

Ele oferece vários templates de sites para anúncio de imóveis, alguns podem ser vistos aqui:

- https://appxperience.imobzi.com/
- https://nexthouse.imobzi.com/
- https://movehouse.imobzi.com/

Oferece vários tipos de pacotes que incluem soluções e valores diferenciados, podendo optar por inserir mais módulos se o cliente optar pelo sistema ERP como podemos ver na Figura 5 (IMOBZI, 2024b).

Figura 5 – Valores dos planos



Fonte: https://www.imobzi.com/planos-para-imobiliarias-e-corretores/

Como pode-se notar na Figura 5, os valores para se manter o sistema no ar são bem altos, sendo um ponto negativo para a escolha do sistema.

Apesar de ser um sistema completo, ele não oferece a opção de personalização dos produtos em casos de alterações na estrutura do banco de dados ou similares. Além disso, o sistema de gestão de locação e o site para imobiliárias, por exemplo, não possuem integração entre si e ainda cobram por usuário ou contrato cadastrado no sistema.

Figura 6 – Plataformas atendidas pelo sistema

Fonte: Seção de vídeos do site: https://sibeleimoveis.com.br

Conforme é possível ver na Figura 6, o sistema de Gestão de Locação da Imobzi atende a todas as plataformas, um ponto positivo para necessidades de gestão que envolvem esta necessidade.

5.4 CONCLUSÃO SOBRE AS FERRAMENTAS

Durante o estudo das ferramentas, conforme a análise, cada uma delas apresenta funcionalidades que atendem, de forma geral, às necessidades identificadas. Essas ferramentas permitem a escolha de modelos e a estilização dos sites, além de serem modularizadas, o que facilita o reuso e se destaca como uma característica positiva.

No entanto, um ponto negativo identificado é a falta de integração desses sites com os softwares de gestão, como contabilidade e controle de caixa. Essa falta de comunicação entre as ferramentas gera complexidade e custos adicionais, além de aumentar os possíveis erros de gestão.

Por outro lado, os sistemas de gestão oferecem muitas funcionalidades que nem sempre são necessárias e acabam cobrando por usuário cadastrado na plataforma, o que pode elevar significativamente os custos de uso. Além disso, esses sistemas não disponibilizam um banco de dados para modificações e não fornecem o código-fonte para adaptação das ferramentas ao contexto do negócio. Também é importante considerar que o uso desses grandes ecossistemas de gestão pode ser mais complexo.

Diante dessas questões, este trabalho tem como objetivo desenvolver um software simples, intuitivo e facilmente integrável, que permita personalização e atenda às reais necessidades da empresa Lar Certo Imóveis, eliminando custos e limitações relacionadas ao cadastro de usuários e contratos.

6 REVISÃO BIBLIOGRÁFICA

Neste tópico foi realizada uma análise das tecnologias utilizadas na implementação da aplicação.

6.1 APLICAÇÃO WEB

Uma aplicação web é um software que utiliza a internet como plataforma para fornecer serviços ou funcionalidades interativas aos usuários. Diferente dos programas tradicionais, que são executados localmente nos dispositivos dos usuários, as aplicações web funcionam em um modelo cliente-servidor, em que o processamento principal ocorre no servidor, enquanto o cliente acessa a aplicação por meio de um navegador (TANENBAUM, 2011).

O modelo cliente-servidor é viável tanto em cenários onde o cliente e o servidor estão próximos fisicamente, como dentro do mesmo prédio, quanto em situações em que estão separados por grandes distâncias. Por exemplo, quando alguém em casa acessa uma página na World Wide Web (WWW), o mesmo modelo é utilizado: o servidor web remoto atua como o servidor, enquanto o computador pessoal do usuário funciona como o cliente (TANENBAUM, 2011).

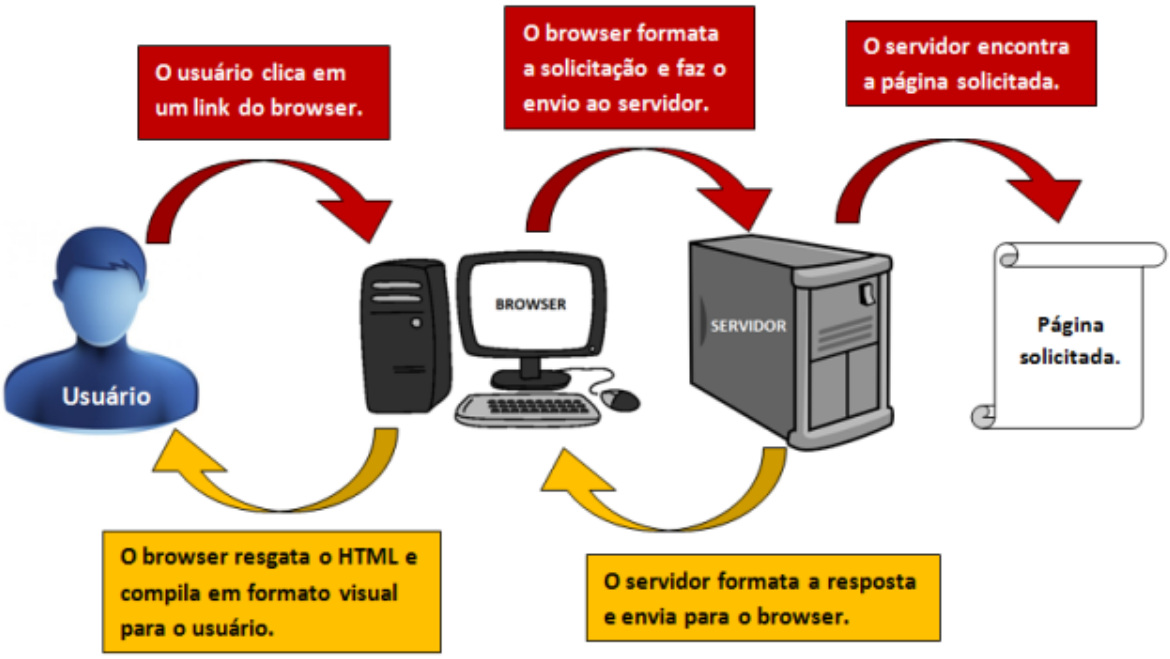
Essa arquitetura permite o acesso remoto, facilitando o uso da aplicação de qualquer lugar com conexão à internet, sem a necessidade de instalação de software adicional, além de centralizar a manutenção e atualizações, garantindo que todos os usuários estejam sempre utilizando a versão mais recente.

Aplicações web são amplamente utilizadas em diversos contextos, como comércio eletrônico, redes sociais e serviços financeiros, oferecendo uma forma prática de interação com funcionalidades complexas.

Na Figura 7 pode se visualizado como um cliente representado pelo browser[1], pode executar uma solicitação ao servidor que irá executar a ação e repassar ao browser que está sendo acessado por um usuário.

[1] Navegador web.

Figura 7 – Fluxo de solicitação e resposta para uma página web



Fonte: DEV MEDIA, 2012

6.2 DESENVOLVIMENTO DE SISTEMA WEB COM REACT

O React é uma biblioteca JavaScript de código aberto amplamente utilizada para criar interfaces de usuário (UI)[2] para aplicativos web e móveis. Ele foi desenvolvido pelo Facebook e lançado em 2013 como uma solução para o problema de atualização de UI em tempo real em aplicativos web, o que o torna uma solução para desenvolvimento web. (REACT, 2024a)

O React é baseado em um modelo de programação declarativo que permite aos desenvolvedores descrever a aparência e o comportamento de uma UI de forma mais simples e intuitiva. Ele também usa um conceito chamado de “componentes” que permite que as partes da UI sejam reutilizadas em diferentes partes do aplicativo, facilitando a manutenção e a escalabilidade do código. (REACT, 2024a)

A componentização é uma abordagem de design em que uma interface de usuário é dividida em pequenos componentes independentes, que podem ser reutilizados em diferentes partes do aplicativo. Cada componente é como uma peça de LEGO[3] que pode ser facilmente encaixada em diferentes partes do aplicativo para construir uma interface de

[2] Parte visual e interativa de um software
[3] Binquedo de peças de plástico encaixáveis que permitem a construção de diversas combinações.

usuário complexa. Isso torna o código mais modular, fácil de entender e manter. (REACT, 2024b)

A reatividade é uma técnica usada para atualizar automaticamente a interface do usuário quando os dados subjacentes mudam. Quando o estado de um componente é alterado, o React detecta automaticamente essas mudanças e atualiza a interface do usuário para refletir as mudanças. Isso é possível graças a um recurso do React chamado Virtual DOM, que permite que o React atualize apenas as partes da interface do usuário que foram alteradas, em vez de atualizar toda a interface do usuário. Isso torna o React extremamente eficiente e rápido. (REACT, 2024b)

6.3 DESENVOLVIMENTO DE SISTEMA BACKEND COM JAVA

Conforme o site da Oracle Java é a linguagem de programação e plataforma de desenvolvimento que reduz custos, encurta os prazos de desenvolvimento, impulsiona a inovação e melhora os serviços de aplicativos. Com milhões de desenvolvedores executando mais de 60 bilhões de Java Virtual Machines(JVM)[4], em todo o mundo, o Java continua a ser a plataforma de desenvolvimento preferida de empresas e desenvolvedores.(ORACLE, 2024)

Java é compilada em bytecode[5] e executada em uma JVM. Isso significa que o código Java pode ser executado em diferentes plataformas, incluindo Windows, Linux e macOS, sem a necessidade de recompilar o código para cada plataforma específica.(BAELDUNG, 2019)

Suporta multithreading[6], o que significa que um programa Java pode executar várias tarefas simultaneamente em threads[7] separadas. Isso é útil para aplicações que precisam executar várias tarefas em paralelo (BAELDUNG, 2022).

É orientada a objetos, o que significa que tudo no Java é um objeto. Isso torna o código Java mais modular e reutilizável, logo é mais fácil de dar manutenção a longo prazo.(BAELDUNG, 2020)

6.4 BANCO DE DADOS RELACIONAL COM POSTEGRESQL

Bancos de dados relacionais são baseados no modelo relacional de dados (DATE 2004). Este modelo foi criado pelo matemático Edgar Frank Codd em 1970, quando era pesquisador da IBM (GOOGLE CLOUD, 2024).

De acordo com Date (2004), um modelo relacional de dados é composto por cinco componentes:

---

[4] Máquina Virtual Java

[5] Código de um programa de computador escrito na linguagem Java é compilado para uma forma intermediária de código denominada bytecode.

[6] Técnica de programação que permite que um programa execute várias tarefas (threads).

[7] Representa um fluxo de execução dentro de um processo.

1) Uma coleção ilimitada de tipos escalares, incluindo o tipo booleano

2) Um gerador de tipo de relação

3) Recursos para definir variáveis de relações

4) Um operador de atribuição relacional

5) Uma coleção ilimitada de operadores relacionais genéricos

O modelo relacional de dados é utilizado para gerenciar informações de forma segura e consistente, com base em regras. Algumas vantagens do modelo relacional são:

- Estrutura organizada, com tabelas e relacionamentos que tornam a organização dos dados intuitiva e eficaz
- Integridade de dados, com o uso de chaves primárias e estrangeiras para evitar inconsistências
- Consultas complexas, com o SQL (Structured Query Language) para recuperar informações específicas de maneira eficiente

O PostgreSQL é um sistema de gerenciamento de banco de dados (SGBD) baseado em modelo relacional que utiliza a linguagem SQL (Structured Query Language) para gerenciar e manipular dados (DEV MEDIA, 2024a). O sistema é uma alternativa gratuita para outros SGBDs comerciais, como o Oracle ou o SQL Server da Microsoft onde se destaca como uma alternativa de código aberto, que oferece robustez e confiabilidade, como evidenciado em testes de estresse, que mostram que o PostgreSQL começa a apresentar falhas nas requisições apenas quando a quantidade de usuários simultâneos ultrapassa 100. Isso não representa um problema, considerando o cenário de poucas requisições ao qual o banco será submetido (POSTGRESQL, 2024).

6.5 QUALIDADE DE SOFTWARE

Segundo Martin (2009) softwares evoluem para a sustentabilidade quando os refatoramos constantemente, auxiliados por testes automatizados, que provém *feedback*[8] rápido sempre que este é alterado. Uma boa cobertura de testes automatizados provê segurança para que o código possa ser modificado por qualquer um, para ser melhorado, re-arquitetado, ou mesmo adaptado.

Em softwares há sempre espaço para melhorias, seja pela simples mudança do nome de uma variável ou de uma função, para torná-las mais expressivas, ou mesmo pela

[8] Feedback é uma palavra inglesa, cujo conceito desenvolve uma adequação no processo de comunicação e resposta.

utilização de técnicas mais sofisticadas para fazerem coisas complexas de formas mais simples e não devemos ter medo de melhorar um código, pelo contrário, temos de ter coragem (Martin, 2009).

Com algumas mudanças no modo em que se pensa sobre o código é possível reduzir consideravelmente o tempo que passamos tentando entender o código existente, sendo assim as refatorizações possibilitam realizar entregas mais eficientes e com maior valor agregado para o cliente.

Os testes unitários permitem que o nosso código seja flexível, sustentável e reutilizável. A razão é simples, se há testes, qualquer modificação no código que quebre uma regra de negócio será instantaneamente informada e passível de ser corrigida! Sem testes, toda mudança é um possível *bug*[9]. Verifica-se que não importa o quão flexível seja sua arquitetura ou o quanto bem particionado seu design, sem testes, um programador ficará relutante em fazer alterações devido ao medo de introduzir bugs não detectados (Martin, 2008).

Segundo Martin (2008), para seguir os princípios do *clean code*[10], algumas boas práticas devem ser adotadas, como no uso de funções, estas devem fazer apenas uma ação e fazê-la bem. Os nomes de variáveis, funções e classes devem ser significativos, descrevendo claramente sua finalidade, evitando a necessidade de comentários, pois um código bem nomeado se explica por si só. Dessa forma a leitura, depuração e teste de código são facilitados.

É importante evitar efeitos colaterais, ou seja, funções que alteram algo fora do seu escopo, bem como eliminar código inutilizado, já que o versionamento garante que nada será perdido. Deve-se evitar repetir código, padronizar nomes para o mesmo conceito e manter o código limpo e organizado. As funções com muitos parâmetros são desaconselhadas, sendo preferível criar objetos para agrupar esses dados. Além disso, os nomes abreviados que dificultam a compreensão devem ser evitados, assim como é necessário manter um espaçamento visual adequado entre os diferentes conceitos no código para facilitar a leitura.

O padrão S.O.L.I.D é um acrônimo que se refere a cinco princípios da programação orientada a objetos, idealizado e construído por Martin (2008). O autor relata sobre a importância desses princípios para escrever código limpo, claro e bem estruturado, que possa ser facilmente mantido e estendido. O padrão S.O.L.I.D é composto pelos seguintes princípios:

- S (Single Responsibility Principle): um objeto deve ter uma única responsabilidade e motivo para mudar.
- O (Open-Closed Principle): uma classe deve estar aberta para extensão, mas fechada para modificação.

[9] Bug é um termo para descrever um comportamento inesperado do sistema.
[10] Clean Code é uma palavra inglesa e sua tradução é código limpo.

- L (Liskov Substitution Principle): uma subclasse deve ser substituível por sua classe base sem alterar o comportamento do programa.
- I (Interface Segregation Principle): muitas interfaces específicas são melhores que uma única interface geral.
- D (Dependency Inversion Principle): os módulos de alto nível não devem depender de módulos de baixo nível, ambos devem depender de abstrações.

Esses princípios ajudam a criar um código mais coeso, flexível, reutilizável e facilmente testável, reduzindo a complexidade e aumentando a qualidade do software.

Portanto, ter um conjunto automatizado de testes unitários que cubram o código de produção é a chave para manter seu design e arquitetura o mais limpo possível. Os testes habilitam todas as *ilities*[11], porque os testes permitem a mudança.

JUnit e Mockito são bibliotecas de teste para Java, que oferecem funcionalidades para facilitar a criação e execução de testes automatizados em projetos Java. Às duas são bibliotecas nativas do Spring Boot.

A JUnit é uma biblioteca de teste que fornece um conjunto de anotações e classes para escrever testes automatizados em Java. Com o JUnit, é possível criar casos de teste para classes e métodos Java, bem como executar e relatar os resultados desses testes (JUNIT, 2024).

O Mockito é uma biblioteca de mock para Java, que permite simular objetos e comportamentos em testes automatizados. Com o Mockito, é possível criar objetos simulados para testar componentes dependentes, como serviços externos, e verificar o comportamento do código testado em diferentes cenários (MOCKITO, 2024).

Originalmente criado pelo Facebook[12], agora Meta[13], o Jest continua a ser desenvolvido e aprimorado por colaboradores de todo o mundo. A biblioteca foi desenvolvida para fornecer uma maneira fácil e eficiente de escrever testes para código JavaScript[14], incluindo testes unitários, de integração e de snapshot[15]. O Jest tem uma sintaxe simples e intuitiva, é altamente configurável e fornece recursos poderosos, como mocking e cobertura de código. Ele também oferece suporte à execução paralela de testes para melhorar o desempenho e à integração com ferramentas de CI/CD para automação de testes contínuos.(JEST, 2024)

Os testes com Jest são paralelizados executando-os em seus próprios processos para maximizar o desempenho, ele pode executar testes em paralelo de maneira confiável. Para tornar as coisas rápidas, ele executa primeiro os testes com falha anterior e reorganiza

[11] Ilities é uma palavra inglesa, e refere-se às características de qualidade de um sistema de software, que terminam com o sufixo “-ilidade” e pode ser utilizado nas palavras técnicas (exemplo): Usability, Maintainability, Scalability, Availability, Extensibility, Portability etc.

[12] Facebook é uma rede social online.

[13] Meta é uma empresa controladora que engloba o Facebook e outras plataformas.

[14] JavaScript é uma linguagem de programação.

[15] Snapshot é uma cópia instantânea do estado de um sistema.

as execuções com base na duração dos arquivos de teste, o que o torna rápido para uso com sistemas de CI/CD (JEST, 2024).

Cabe também salientar ser uma dependência de desenvolvimento, isso significa que ele não é necessário para a execução do aplicativo em produção, mas sim, para garantir a qualidade do código e evitar problemas durante o desenvolvimento (JEST, 2024).

6.6 CI/CD

O CI/CD significa integração contínua e entrega contínua (em inglês, *Continuous Integration/Continuous Delivery*). É uma prática de desenvolvimento de software que visa a automatização de todo o processo de construção, testes e distribuição do software. A integração contínua consiste em integrar o código dos desenvolvedores ao repositório compartilhado frequentemente e realizar testes automatizados para verificar a qualidade do código. Já a entrega contínua envolve a automação do processo de liberação do software, com a possibilidade de implantação em produção de forma rápida e segura (FOWLER, 2010).

Um sistema de versionamento, também conhecido como controle de versão ou VCS (*Version Control System*), é um software que gerencia mudanças em arquivos e diretórios ao longo do tempo, permitindo que várias pessoas trabalhem em um projeto de software simultaneamente e mantenham um histórico das alterações realizadas.

Seguindo padrões como o *Semantic Versioning* (MAJOR.MINOR.PATCH, ex.: 1.0.0, 2.1.3), o GitHub oferece uma forma de marcar uma versão específica do código no repositório por meio da criação de uma *tag*. Uma tag é muito similar a uma branch, mas segue o princípio da *imutabilidade*: uma vez criada, ela não pode ser alterada, o que a torna uma ferramenta confiável para fins de rastreabilidade e identificação de versões (SEMVER, 2024).

Este projeto adotou a semântica de versionamento, com as seguintes definições:

- *MAJOR* (*grande mudança*): usado quando há alterações incompatíveis na API.
- *MINOR* (*nova funcionalidade*): usado para adicionar funcionalidades de forma compatível com versões anteriores.
- *PATCH* (*correção*): usado para aplicar correções de bugs que mantêm a compatibilidade com versões anteriores.

Os sistemas de versionamento também permitem que os usuários recuperem versões anteriores de um arquivo, comparem alterações ao longo do tempo e revertam alterações específicas, caso necessário (GIT, 2014).

O GitHub é uma plataforma *web* de hospedagem de código-fonte e colaboração em projetos de software. Ele permite que os desenvolvedores armazenem e compartilhem seu código-fonte, além de colaborar em projetos com outros desenvolvedores ele disponibiliza um amplo ecossistema de desenvolvimento. O GitHub pode ser utilizado por desenvolvedores para hospedar e colaborar em projetos de código aberto, mas também pode ser usado para projetos privados e comerciais. As principais funcionalidades do GitHub incluem:

Repositórios: permitem armazenar e gerenciar o código-fonte do projeto, assim como controlar as diferentes versões do código por meio do Git.

Issues: permitem criar e acompanhar tarefas, bugs e outras demandas relacionadas ao projeto. É possível adicionar descrições, atribuir responsáveis, definir prioridades, entre outras informações.

Pull Requests: permitem propor alterações ao código-fonte do projeto e discutir essas alterações com outros colaboradores antes de integrá-las ao código principal.

Wikis: permitem criar e editar documentação do projeto, como guias de instalação, tutoriais e outros documentos relevantes.

Projetos: permitem criar quadros de tarefas para gerenciar o progresso do projeto, seguindo a metodologia Kanban ou outra metodologia de gerenciamento de projetos.

Integrations: permitem integrar o GitHub com outras ferramentas e serviços, como sistemas de integração contínua, plataformas de hospedagem de aplicativos, entre outros (GITHUB, 2024).

6.7 METODOLOGIA ÁGIL

Segundo Corrêa (2017), ele define metodologia ágil como um conjunto de práticas, valores e princípios que buscam a melhoria contínua do processo de desenvolvimento de software, visando à satisfação do cliente e a entrega de um produto de qualidade.

Segundo o autor, a metodologia ágil preconiza a flexibilidade, a adaptabilidade, a comunicação e a colaboração entre as equipes envolvidas no projeto, além de uma maior interação com o cliente, que é considerado parte integrante do processo de desenvolvimento. O artigo apresenta uma revisão bibliográfica sobre o tema, bem como um estudo de caso sobre a adoção da metodologia ágil em uma empresa de desenvolvimento de software.

Segundo o autor, tanto o Scrum quanto o Kanban são metodologias ágeis que visam a melhoria contínua do processo de desenvolvimento de software e a entrega de valor ao cliente de forma rápida e constante. Ambas as metodologias são baseadas em princípios como a colaboração, a adaptação, a auto-organização e a entrega contínua.

No entanto, o autor destaca que o Scrum é uma metodologia mais estruturada e prescritiva, com papéis, cerimônias e artefatos bem definidos, enquanto o Kanban é mais flexível e adaptável, com foco na melhoria contínua e na visualização do fluxo de trabalho.

O autor também destaca que tanto o Scrum quanto o Kanban podem ser aplicados

em diferentes contextos e situações, e que a escolha da metodologia mais adequada depende das características do projeto, da equipe e do cliente envolvido (CORRÊA, 2017).

O Kanban é uma forma holística de pensar sobre seus serviços com o foco em melhorá-los a partir da perspectiva de seus clientes, podendo ser utilizado para se empregar uma metodologia ágil.

> Trata-se, antes, de um método ou de uma abordagem de gestão que deve ser aplicada a um processo ou método de trabalho já existente(KANBAN, 2021, p4)

Conforme consta no guia oficial do método Kanban (KANBAN, 2021), o método se baseia em algumas práticas fundamentais para ajudar a organizar e melhorar o trabalho em equipe.

É preciso visualizar como o trabalho flui, mapeando cada etapa para identificar gargalos e desperdícios. Após, é importante limitar a quantidade de trabalho em andamento ao mesmo tempo, evitando sobrecarregar o desenvolvimento e garantindo que o trabalho seja concluído antes de iniciar novas tarefas. A ideia também é gerenciar continuamente o fluxo para que tudo flua bem, resolvendo problemas que possam surgir.

Além disso, as regras e políticas do processo precisam estar bem claras, o que ajuda a saber o que fazer e como fazer. O feedback constante é fundamental, pois permite que a equipe faça ajustes e melhore a eficiência. E, por fim, o Kanban incentiva melhorias contínuas, incentivando a equipe a colaborar e experimentar novas maneiras de otimizar o processo.

7 METODOLOGIA

O capítulo a seguir explica de forma detalhada como foram realizadas cada uma das etapas da metodologia.

7.1 PESQUISA

Este trabalho pode ser classificado, quanto à abordagem, como pesquisa qualitativa, ou seja, ele se preocupa “com o aprofundamento da compreensão de um grupo social, de uma organização, etc.” (GERHARDT e SILVEIRA, 2009, p. 31).

Em geral, a pesquisa qualitativa utiliza métodos de coleta de dados que permitem a obtenção de informações descritivas e subjetivas, tais como entrevistas, observação participante e análise de documentos (Flick, 2018).

Nesta perspectiva, foi realizada uma entrevista com o usuário do software representativo do sistema que foi desenvolvido. A entrevista dispõe de um roteiro com perguntas abertas que permitiram ao usuário expressar suas opiniões e necessidades em relação ao software. Assim, podemos classificar a metodologia como uma pesquisa aplicada, pois os dados levantados foram utilizados em uma solução prática.

Por conseguinte, classifica-se como pesquisa exploratória, pois foi realizado o levantamento de informações a respeito de um problema a partir de entrevistas com o usuário final do sistema (GIL, 2019).

7.2 ANÁLISE DE REQUISITOS

Inicialmente, para este trabalho, foi realizado uma pesquisa exploratória em publicações, documentações, notícias e artigos para que se possa construir o referencial teórico. Uma pesquisa em sistemas semelhantes ao desenvolvido neste trabalho também foi realizada, com objetivo de estruturar os requisitos e definir as funcionalidades que serão implementadas para solucionar o problema apresentado.

Também foi realizada uma análise para escolha das melhores tecnologias que serviram de base no desenvolvimento da aplicação web durante todo o processo de desenvolvimento.

7.3 MODELAGEM

Para descrever os fluxos do sistema e suas funcionalidades principais foram gerados diagramas (UML Linguagem de Modelagem Unificada) e ER (Entidade-Relacionamento). Tendo em vista a escolha de banco relacional, o modelo ER é uma técnica usada para modelar dados em um nível conceitual, e é amplamente utilizado para projetar bancos

de dados relacionais. O UML é uma grande aliada na implementação e planejamento de sistemas tanto estruturalmente quanto para comportamentos (LUCIDCHART, 2021).

7.4 IMPLEMENTAÇÃO

A linguagem de programação utilizada para o desenvolvimento da aplicação é o Java. Esta linguagem utiliza o paradigma orientado a objetos, onde todos os elementos inseridos são objetos que interagem entre si. Uma das grandes vantagens do Java é sua grande comunidade, tornando mais simples resolver problemas que possam ocorrer durante a codificação de um sistema (ORACLE, 2024).

O framework utilizado no desenvolvimento é o Spring Boot, que oferece facilidade e flexibilidade para trabalhar com requisições Web. Os muitos recursos criados especificamente para o Spring Boot facilitam a criação e a execução de seus microsserviços em produção em escala que são uma abordagem moderna de software em que o código do aplicativo é entregue em partes pequenas e gerenciáveis e independentes umas das outras (SPRING, 2024).

8 DESENVOLVIMENTO

Este capítulo apresenta o desenvolvimento prático do projeto, como a construção da aplicação, aprofundamento nas tecnologias, camadas e processos técnicos.

8.1 ANÁLISE DE REQUISITOS

Para definir as funcionalidades essenciais do sistema de gestão de aluguéis, foi realizado uma análise de requisitos inicial por meio de entrevista com a principal stakeholder do projeto, que também é a proprietária da empresa de gestão imobiliária. Durante a entrevista, foram identificados pontos importantes sobre a operação atual e as necessidades específicas para o novo sistema.

Conforme relatado pela stakeholder, o gerenciamento dos imóveis e contratos de aluguel era realizado exclusivamente em planilhas de Excel, com foco no controle de imóveis alugados ou disponíveis. No entanto, a ferramenta atual não oferecia suporte para geração de relatórios de demonstrativos financeiros detalhados, como os de gastos operacionais e rendimentos dos imóveis, dificultando uma visão clara dos custos e receitas da empresa.

Com base na análise inicial dos processos da empresa, foram identificados e definidos os principais requisitos visando informatizar e automatizar as operações essenciais para a gestão imobiliária. Além disso, foram incluídas novas funcionalidades projetadas para aprimorar o controle financeiro e a administração de aluguéis. Esses requisitos foram organizados em uma lista detalhada, disponível neste documento compartilhado[1] e serviram como base para o desenvolvimento das funcionalidades principais do sistema. Entre essas funcionalidades destacam-se:

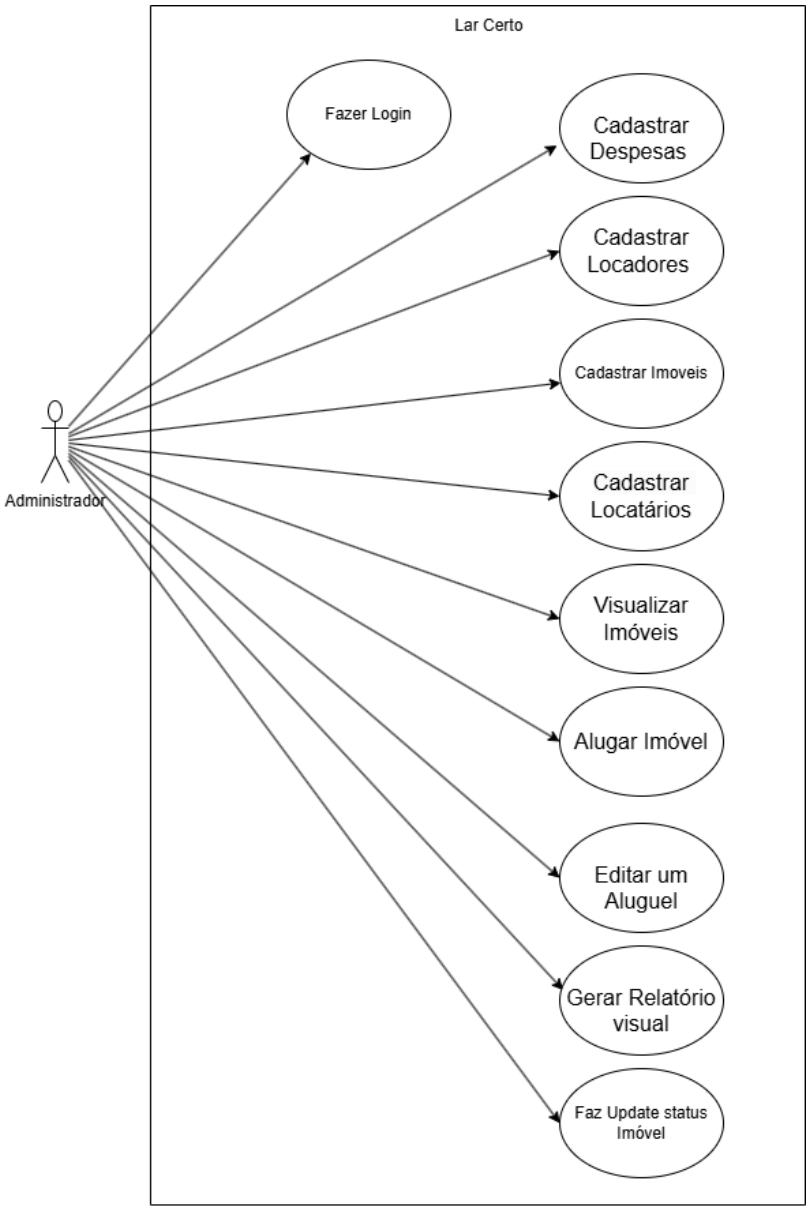
- Cadastro de Usuários do Sistema: controle de acesso e criação de perfis de usuário para funcionários ou administradores, garantindo a segurança e personalização do uso;
- Gerenciamento de Imóveis: cadastro e organização de informações sobre cada imóvel, incluindo status de disponibilidade (alugado ou não), com atualização em tempo real;
- Controle de Aluguéis: registro e monitoramento de contratos de aluguel, com informações detalhadas sobre o período do contrato, valores acordados e histórico de ocupação;
- Cadastro de Locatários e Inquilinos: centralização de dados dos proprietários (locatários) e dos inquilinos, incluindo histórico de relacionamento com a empresa;

[1] Documento disponível em: https://docs.google.com/document/d/15nNna5BLgZEy52mKH1IroD1l4mv8cYegrBoUrFT_jAs/edit?usp=sharing. Acesso em:24 nov 2024.

- Relatórios e Gráficos de Custos e Rendimentos: implementação de uma funcionalidade de relatórios financeiros, com gráficos detalhados sobre custos operacionais e rendimentos obtidos pelos imóveis. Esses relatórios possibilitam uma análise mais precisa e acessível dos dados financeiros da empresa, auxiliando na tomada de decisões estratégicas.

Com base nesses requisitos foi gerado um diagrama de caso de uso conforme demonstrado na figura 8 e o diagrama de classe representado pela figura 9.

Figura 8 – Diagrama de caso de uso



Fonte: Autoria própria (2024)

Para melhor visibilidade do diagrama de classes, ele está disponível em link[2] público,

[2] Diagrama de Classe, acessível em: https://drive.google.com/file/d/1RIt3OtRytrgx2ItjZ7Ruf7pDqr4ff3tH/view

e mencionado na referência bibliográfica (SOUZA, 2024) .

Essa análise de requisitos, conduzida com a participação ativa da stakeholder, garantiu que o sistema fosse desenvolvido conforme as reais necessidades do negócio, substituindo as planilhas de Excel por uma plataforma robusta e eficaz para a gestão imobiliária. A implementação dos relatórios gráficos financeiros foi especialmente importante, pois agregou valor ao sistema, proporcionando à empresa uma visão completa e detalhada de seu desempenho financeiro, algo antes inexistente.

8.2 ORGANIZAÇÃO E PLANEJAMENTO TÉCNICO

Para iniciar o desenvolvimento técnico do projeto, adotou-se o método Kanban como ferramenta de organização e controle de tarefas, onde me utilizei de um quadro na ferramenta online Miro (MIRO, 2024). A metodologia Kanban junto ao uso do Miro permitiu visualizar o fluxo de trabalho, facilitando o gerenciamento de atividades e a identificação de prioridades e possíveis gargalos ao longo do desenvolvimento. As etapas iniciais foram planejadas para a construção de um Produto Mínimo Viável (MVP), focando nas funcionalidades essenciais para o sistema de gestão de aluguéis.

O desenvolvimento foi dividido em diferentes etapas que incluíram o planejamento do MVP, organização do repositório, configuração do ambiente de desenvolvimento e implantação de uma pipeline de CI/CD (Integração Contínua e Entrega Contínua), conforme detalhado a seguir.

Planejamento do MVP: Através do Kanban, as principais funcionalidades definidas na análise de requisitos foram convertidas em tarefas individuais, categorizados em três tipos: *essenciais, importantes e desejáveis* e organizadas em colunas de progresso (“A fazer”, “Em andamento” e “Concluído”).

Os requisitos *essenciais* foram definidos como aqueles que deveriam necessariamente ser desenvolvidos para garantir a funcionalidade mínima do sistema. Os requisitos *importantes* foram considerados uma prioridade intermediária, sendo aqueles que teriam impacto significativo na qualidade e eficiência do sistema, mas que não eram obrigatórios para seu funcionamento básico. Já os requisitos *desejáveis* seriam implementados caso houvesse tempo suficiente, adicionando valor ao sistema, mas sem comprometer sua funcionalidade em caso de ausência. A utilização dessa estrutura possibilitou uma visão clara do avanço do projeto e uma gestão mais eficiente do desenvolvimento, pois facilitou a priorização de tarefas ao longo do processo.

Organização do Repositório no GitHub: Criou-se um repositório específico para o projeto, onde todos os arquivos do código-fonte foram centralizados. A utilização do GitHub permitiu a rastreabilidade das alterações, garantindo o versionamento seguro do código e a colaboração eficiente entre os envolvidos.

?usp=sharing

Configuração de CI/CD: Para otimizar o fluxo de desenvolvimento e garantir a qualidade das entregas, implementou-se uma pipeline de CI/CD no GitHub. Utilizando o GitHub Actions(seu uso será detalhado mais a frente), configurou-se uma série de ações automatizadas que incluíam o build do projeto, execução de testes automatizados e a criação de uma imagem Docker para implantação. Essa configuração de CI/CD automatizou etapas importantes, permitindo que cada atualização no repositório fosse validada e preparada para o ambiente de produção de forma rápida e segura.

As histórias de usuário desenvolvidas para este projeto, que descrevem as funcionalidades e requisitos do sistema de forma clara e orientada às necessidades dos usuários, estão organizadas no *Apêndice A*. Elas foram elaboradas para guiar o desenvolvimento do sistema e garantir que as entregas estejam alinhadas às expectativas e objetivos definidos no planejamento do projeto.

Com essa organização inicial, o trabalho técnico do projeto manteve-se estruturado e eficiente, com controle visual das tarefas e automação do processo de entrega, elementos que contribuíram diretamente para a qualidade e consistência do desenvolvimento do sistema de gestão de aluguéis.

Dada a fase inicial de planejamento do projeto e com o objetivo de construir um Produto Mínimo Viável (MVP), optei por uma arquitetura simplificada do tipo MVC (Model-View-Controller) na API. Essa escolha tem base na necessidade de uma estrutura de fácil implementação e manutenção, que permitisse desenvolver rapidamente as funcionalidades essenciais e, ao mesmo tempo, garantisse uma organização clara entre as camadas de dados, lógica e apresentação.

Para complementar a arquitetura, o desenvolvimento incluiu um front-end em React, proporcionando uma interface dinâmica e interativa para os usuários finais. O uso do React possibilitou uma integração eficiente com a API, assegurando uma troca de dados fluida e responsiva.

A escolha pela arquitetura MVC foi guiada por três fatores principais:

Separação de Responsabilidades: A estrutura MVC, aliada à divisão em pacotes model, service e repository, facilita a organização do código, separando as camadas de dados, lógica e apresentação. Essa organização modular melhora a leitura e manutenção do código, tornando-o ideal para o desenvolvimento iterativo e escalável do MVP.

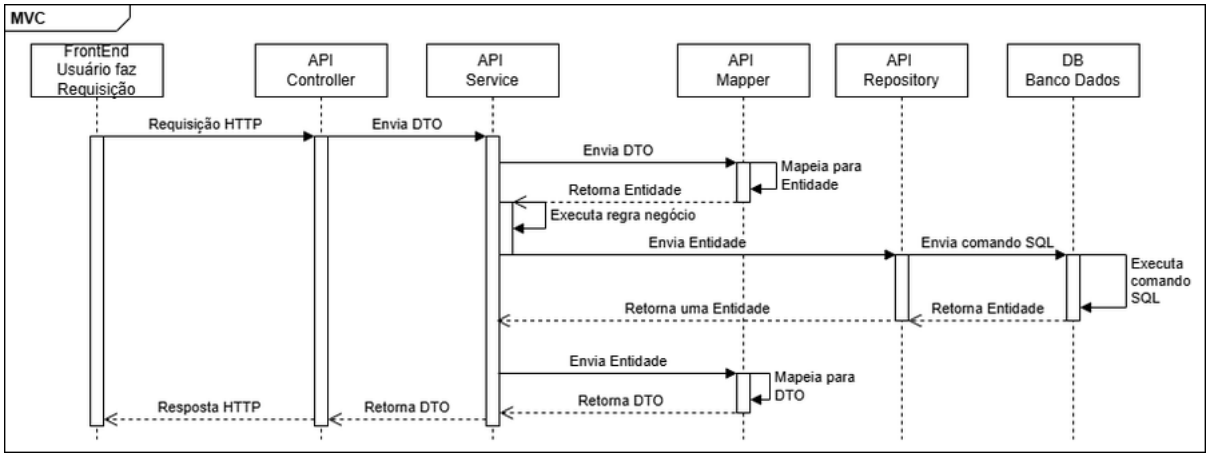
Facilidade de Expansão: Com foco inicial nas funcionalidades principais, a arquitetura MVC permite expandir o sistema sem comprometer sua organização. Essa base modular é vantajosa caso novos requisitos sejam incorporados no futuro.

Integração com o Spring Boot e o React: O Spring Boot oferece suporte nativo para o padrão MVC, simplificando a implementação de cada camada. Além disso, o uso do React no front-end proporcionou uma interface rica e interativa para o usuário, integrando-se bem à API REST desenvolvida.

Dessa forma, a combinação de uma arquitetura MVC com a organização em pacotes

model, service e repository, e o uso do React no front-end, adequou-se ao objetivo de desenvolver um MVP ágil e funcional, permitindo um sistema estável e escalável para as futuras fases de desenvolvimento.

Figura 9 – Fluxo Dados na Arquitetura MVC



Fonte: autoria própria (2024)

Na figura 9 é demonstrado como o fluxo ocorre na aplicação, repare que foi utilizado o conceito de DTO, um padrão de design utilizado para transferir dados entre diferentes partes de uma aplicação ou entre aplicações distintas. Seu principal objetivo é encapsular dados que precisam ser transportados de um sistema para outro (por exemplo, entre a camada de serviço e a interface do usuário), simplificando e otimizando a comunicação.

8.3 DOCUMENTAÇÃO DA API COM USO DO SWAGGER

O Swagger é uma ferramenta amplamente utilizada para a documentação de APIs REST. Ele permite que os desenvolvedores documentem, testem e interajam com APIs de forma simples e eficiente. O Swagger facilita o entendimento da API tanto para desenvolvedores como para outras partes interessadas, garantindo que a estrutura e os detalhes dos endpoints estejam claros e acessíveis.(SWAGGER, 2024a)

8.3.1 O que é Swagger?

O Swagger é um conjunto de ferramentas que automatiza a geração de documentações interativas para APIs RESTful. Utilizando a especificação OpenAPI, o Swagger fornece uma interface visual onde os usuários podem testar os endpoints da API diretamente a partir do navegador, sem a necessidade de ferramentas externas, como o Postman ou CURL.

A principal funcionalidade do Swagger é converter as definições da API em uma interface de usuário amigável e interativa, onde é possível visualizar todos os detalhes de cada endpoint, como:

- Métodos HTTP (GET, POST, PUT, DELETE)
- Parâmetros de entrada (query, path, body, header)
- Respostas possíveis (status code, formato de resposta)
- Descrição dos recursos e exemplos

8.3.2 Vantagens de Usar o Swagger

A adoção do Swagger traz várias vantagens para o desenvolvimento e a manutenção de APIs(SWAGGER, 2024b), entre as quais se destacam:

Documentação Automática: A partir das anotações no código, o Swagger gera automaticamente a documentação, reduzindo a necessidade de documentação manual.

Teste Interativo: Com o Swagger UI, é possível testar os endpoints diretamente na interface gráfica, facilitando a validação e o desenvolvimento.

Especificação Padronizada: A documentação segue a especificação OpenAPI, garantindo que a API esteja de acordo com os padrões globais de descrição de APIs.

Facilidade de Integração: Outras equipes ou sistemas podem entender e integrar-se rapidamente à API, graças à clareza da documentação fornecida pelo Swagger.

8.3.3 Como o Swagger foi Utilizado

Na implementação da API deste projeto, o Swagger foi utilizado para documentar e testar os endpoints de forma prática e eficiente. O Swagger UI foi habilitado para permitir o acesso à documentação diretamente no navegador, utilizando as anotações fornecidas pelo Spring Boot.(SPRINGDOC, 2024)

- Configuração Básica
- A configuração do Swagger no projeto foi feita através das dependências do Springdoc OpenAPI, que é amplamente usado em projetos Spring Boot. Abaixo está um exemplo de configuração simples:

Primeiro é necessário a importação da dependência no arquivo build.gradle.kts, onde segue o trecho de código necessário a ser inserido:

```
1 implementation("org.springdoc:springdoc-openapi-starter-webmvc-ui:2.3.0")
```

- Após adicionar a dependência, a documentação da API já fica acessível automaticamente no endereço:

- **http://localhost:8080/swagger-ui.html**

Como nesta API foi utilizado autenticação via token JWT (JSON Web Token), foi necessário configurar o swagger para que ele utilize o token nas requisições e nesta mesma configuração é realizada a inserção das demais informações do sistema como demonstrado a seguir, criando um objeto do tipo OpenAPI:

```
1 @Configuration
2 public class SwaggerConfig {
3
4   private static final String BEARER_AUTH = "Bearer ";
5
6   @Bean
7   public OpenAPI testOpenAPIDefinition() {
8
9     return new OpenAPI()
10        .info(new io.swagger.v3.oas.models.info.Info()
11            .title("API - Lar Certo Imóveis")
12            .contact(new Contact()
13                .name("Anderson Fuhr").email("andersonfuhr@yahoo.com.br"))
14            .description("Aplicação para controle de imóveis")
15            .version("v0.0.1"))
16        .externalDocs(new ExternalDocumentation()
17            .description("Repositório Desta API no GitHub")
18            .url("https://github.com/fuhr-br"))
19        .addSecurityItem(new SecurityRequirement()
20            .addList(BEARER_AUTH))
21        .components(new Components()
22            .addSecuritySchemes(BEARER_AUTH, new SecurityScheme()
23                .name(BEARER_AUTH)
24                .type(SecurityScheme.Type.HTTP)
25                .scheme("bearer")
26                .bearerFormat("JWT")));
27   }
28
29 }
```

A configuração do código acima trabalha com o *design pattern* builder, onde cada método recebe uma entrada e retorna um objeto, se não o mesmo objeto, onde é possível configurar um objeto do tipo OpenAPI. Abaixo segue a descrição do significado da configuração:

Então é necessário setar as configurações das Informações básicas da API como demonstrado a seguir:

```
1 .title("API - Lar Certo Imóveis")
2 .contact(new Contact()
3     .name("Anderson Fuhr").email("andersonfuhr@yahoo.com.br"))
4 .description("Aplicação para controle de imóveis")
5 .version("v0.0.1"))
```

title: O título da API.
contact: Informações de contato do autor da API.
description: Uma breve descrição do que a API faz.
version: A versão atual da API.

Após pode ser informado o link contendo uma documentação externa do sistema, neste caso foi utilizado a url do repositório do github como exemplo a seguir:

```
1 .externalDocs(new ExternalDocumentation()
2     .description("Repositório Desta API no GitHub")
3     .url("https://github.com/fuhr-br"))
```

externalDocs: Adiciona documentação externa, como um link para o repositório do GitHub, onde os usuários podem encontrar mais informações.

Com as configurações básicas inseridas, pode ser informado ao swagger que o sistema usará algum tipo de autenticação de segurança como mostra no código abaixo:

```
1 .addSecurityItem(new SecurityRequirement()
2     .addList(BEARER_AUTH))
```

Então por fim, se define que a API requer autenticação baseada em bearer token, que será fornecido no cabeçalho das solicitações como demonstrado no trecho abaixo:

```
1 .components(new Components()
2     .addSecuritySchemes(BEARER_AUTH, new SecurityScheme()
3         .name(BEARER_AUTH)
4         .type(SecurityScheme.Type.HTTP)
5         .scheme("bearer")
6         .bearerFormat("JWT")));
```

components: Este bloco é onde você define os esquemas de segurança.
addSecuritySchemes: Adiciona o esquema de segurança definido anteriormente.
name: Nome do esquema de segurança.
type: Tipo de segurança (neste caso, HTTP).
scheme: O esquema de autenticação (neste caso, “bearer”).
bearerFormat: O formato do token (neste caso, JWT).

8.3.4 Anotações Utilizadas

No código da API, as anotações @Operation e @ApiResponse foram usadas para fornecer detalhes específicos de cada endpoint, incluindo uma breve descrição e os possíveis códigos de resposta. Veja um exemplo de anotação para documentar o cadastro de um novo aluguel:

```
1 @PostMapping
2 @Operation(summary = "Cadastra um novo Aluguel", description = "Endpoint
      acessível apenas para usuários com perfil de ADMINISTRADOR.")
3 @ApiResponses(value = {
4     @ApiResponse(responseCode = "201", description = "Aluguel cadastrado com
          sucesso"),
5     @ApiResponse(responseCode = "403", description = "Acesso negado")
6 })
7 public ResponseEntity< AluguelResponse > salvar(@RequestBody @Validated
      AluguelRequest aluguelRequest) {
8     return ResponseEntity.status(HttpStatus.CREATED).body(service.salvar(
          aluguelRequest));
9 }
```

Essa anotação gera automaticamente a documentação deste endpoint, descrevendo sua função (cadastrar um novo aluguel), os possíveis códigos de resposta (201 para sucesso e 403 para acesso negado), além de permitir o teste direto na interface do Swagger UI (SPRINGDOC, 2024).

8.3.5 Teste Interativo com Swagger UI

Uma das grandes vantagens do Swagger é a possibilidade de testar a API diretamente da interface de documentação. Após acessar a interface no endereço /swagger-ui.html, é possível:

- Ver todos os endpoints disponíveis, organizados de maneira clara.
- Inserir dados nos parâmetros dos métodos (como corpo da requisição, parâmetros de URL ou cabeçalhos).
- Executar chamadas HTTP diretamente pela interface para verificar as respostas da API.

Assim, facilita muito o desenvolvimento como o processo de QA (Quality Assurance), já que todos os testes de integração funcionais podem ser feitos diretamente a partir do Swagger, sem a necessidade de uso de ferramentas externas.

O uso do Swagger foi fundamental para garantir a documentação clara e interativa dos endpoints da API. Ele proporcionou uma maneira simples de visualizar e testar a API durante o desenvolvimento, além de facilitar a integração com outras partes interessadas. Por meio da geração automática de documentação, o Swagger eliminou a necessidade de manutenção manual, garantindo que a documentação estivesse sempre atualizada e sincronizada com a API.

A documentação fornecida pelo Swagger garantiu que a API fosse de fácil compreensão e utilização, permitindo uma interação transparente com os recursos da aplicação e melhorando a colaboração entre as futuras equipes envolvidas no projeto.

8.4 IMPLEMENTAÇÃO DA CAMADA DE LOGIN COM SPRING SECURITY

No artigo escrito pelo autor desse projeto no Linkedin (Souza, 2024), o Spring Security facilita a implementação de segurança em aplicação Java, fornecendo autenticação e proteção contra ameaças comuns da web.

A camada de segurança é uma parte essencial em aplicações web modernas, garantindo que apenas usuários autorizados tenham acesso a determinadas funcionalidades. Neste capítulo, abordarei a construção da camada de login, utilizando o framework Spring Security, que oferece ferramentas robustas para autenticação e autorização em aplicação Java. O objetivo deste capítulo é descrever o processo de implementação dessa camada e as principais funcionalidades empregadas para garantir a segurança da aplicação.

Neste trabalho, foi utilizado o protocolo de autorização OAuth 2.0, que permite que aplicativos acessem recursos protegidos em nome de um usuário sem expor suas credenciais. O OAuth 2.0 oferece diversos fluxos de autorização adaptados a diferentes cenários, como:

Authorization Code Flow: O usuário é autenticado, gerando um código de autorização que permite a obtenção de um token de acesso à aplicação.

Client Credentials Flow: Utilizado quando um sistema consome diretamente um recurso, validando as credenciais da aplicação.

Uma das vantagens do OAuth 2.0 é a possibilidade de separar a autenticação em um serviço independente, como um microserviço. No entanto, este projeto foi implementado como um monolito para otimizar o uso de infraestrutura.

O token JWT para acesso à aplicação é um padrão aberto definido pela RFC 7519 (Request for Comments), que define uma maneira compacta e autocontida de transmitir informações com segurança entre as partes como um objeto JSON. Essas informações podem ser verificadas e confiáveis porque são assinadas digitalmente. JWTs podem ser assinados usando um segredo ou um par de chaves pública/privada.(7519, 2024)

Os JWTs são úteis em cenários de autorização, onde, após o login, cada solicitação do usuário inclui o token para acessar recursos permitidos, sendo amplamente utilizado em Single Sign-On devido à sua leveza e fácil uso entre domínios. Além disso, eles são eficazes para troca segura de informações, pois podem ser assinados digitalmente, garantindo a autenticidade e integridade dos dados, motivo pelo qual optei pelo uso no desenvolvimento do trabalho (JWT.IO, 2024).

8.4.1 Visão Geral do Spring Security

O Spring Security é um framework de segurança poderoso para aplicações Java, amplamente utilizado em conjunto com o Spring Boot. Ele oferece uma série de funcionalidades, incluindo autenticação de usuários, autorização baseada em perfis, proteção

contra ameaças comuns da web (como ataques CSRF e XSS), além de uma fácil integração com APIs RESTful, através de autenticação baseada em tokens JWT (SPRING SECURITY, 2024).

A escolha do Spring Security para este projeto foi motivada pela sua flexibilidade e ampla adoção na comunidade Java, além de sua integração nativa com o ecossistema Spring.

8.4.2 Estrutura da Autenticação e Autorização

A construção da camada de login foi realizada utilizando os mecanismos de autenticação e autorização do Spring Security. Para a autenticação, foi implementado um sistema que gera tokens JWT para identificar os usuários de forma segura e eficiente. A autorização foi baseada em perfis de usuários, como ADMINISTRADOR e USUARIO, controlando o acesso a recursos específicos da aplicação.

8.4.3 Cadastro de Usuário

O sistema permite o cadastro de novos usuários através de um endpoint público de registro, onde o primeiro usuário pode ser criado com o perfil de ADMINISTRADOR. Em ambientes de produção, esse endpoint é restrito, garantindo que apenas administradores possam registrar novos usuários. O código responsável pelo registro está logo abaixo:

```
1  @PostMapping("/registrar")
2      @Operation(summary = "Cadastra um novo usuário", description = "Acessível
           somente para ADMINISTRADOR")
3      public ResponseEntity< String > registrar(@RequestBody @Valid RegistroDTO
           registroDTO) {
4          autorizacaoService.cadastrar(registroDTO);
5          return ResponseEntity.status(HttpStatus.CREATED).body("Usuario
               cadastrado com sucesso");
6      }
```

8.4.4 Processo de Login

O login é realizado por meio de um endpoint que recebe as credenciais do usuário e, se forem válidas, gera um token JWT. Este token é utilizado nas próximas requisições para identificar o usuário e autorizar seu acesso. A seguir, o código que implementa o login na API:

```
1      @PostMapping("/login")
2      @Operation(
3            summary = "Faz login de um usuário cadastrado no sistema",
4            description = "Acessível para qualquer usuário"
5      )
6      public ResponseEntity< TokenDTO > login(@RequestBody @Validated
           AutenticacaoDTO autenticacaoDTO) {
```

```
7         return ResponseEntity.ok(autorizacaoService.login(autenticacaoDTO));
8         }
```

Esse token JWT contém informações como o nome de usuário e suas roles, além de uma data de expiração, garantindo a segurança e validade temporária do acesso.

No front end, o sistema pode então extrair as informações expostas no token, e utilizar esses dados para restringir acessos aos componentes aos quais o usuário pode ou não acessar.

O código abaixo implementa um provider de autenticação para gerenciar o estado de autenticação de um usuário usando JWT. Ele utiliza o React Context API(REACT, 2024c) para fornecer funções de login, logout e verificação de autenticação para o resto da aplicação.

```
1  export const AuthProvider = ({ children }) => {
2    const [authenticated, setAuthenticated] = useState(false);
3    const tokenEstaExpirado = (token) => {
4      try {
5        const tokenPayload = JSON.parse(atob(token.split('.')[1]));
6        const expirationDateString = tokenPayload.expiration;
7        const expirationDate = new Date(expirationDateString);
8        const currentDate = new Date();
9        return expirationDate < currentDate;
10     } catch (error) {
11       return true;
12     }
13   };
14   useEffect(() => {
15     const token = localStorage.getItem(userTokenName);
16     if (token) {
17       if (tokenEstaExpirado(token)) {
18         localStorage.removeItem(userTokenName);
19         setAuthenticated(false);
20       } else {
21         setAuthenticated(true);
22       }
23     }
24   }, []);
25   const login = (token) => {
26     localStorage.setItem(userTokenName, token);
27     setAuthenticated(true);
28   };
29   const logout = () => {
30     localStorage.removeItem(userTokenName);
31     setAuthenticated(false);
32   };
33   const isAuthenticated = () => {
34     return authenticated;
35   };
36   return ({children});
37 };
```

- *useState*: Mantém o estado local authenticated, que indica se o usuário está autenticado.
- *tokenEstaExpirado*: Função que verifica se o token JWT armazenado no navegador está expirado. Ela faz isso decodificando a carga útil do token (payload) usando

atob() para converter a parte codificada do token (base64) e, em seguida, compara a data de expiração do token com a data atual.

  - Se a data de expiração for anterior à data atual, o token é considerado expirado.
  - Em caso de erro na decodificação (token inválido), também é tratado como expirado.

- *useEffect*: Quando o componente é montado, o useEffect verifica se há um token armazenado no localStorage. Se existir:

  - Verifica se o token está expirado.
  - Se estiver, remove o token e define o estado authenticated como false.
  - Se não estiver, o usuário é considerado autenticado e authenticated é definido como true.

- *login(token*): Esta função salva o token no localStorage quando o usuário faz login e define authenticated como true, indicando que o usuário está logado.

- *logout():* Remove o token do localStorage e define authenticated como false, desconectando o usuário.

- *isAuthenticated():* Retorna o valor atual de authenticated, permitindo que outros componentes da aplicação saibam se o usuário está logado.

Na aplicação React, a configuração da camada de login com o JWT foi estendida para controlar as rotas, garantindo que apenas usuários autenticados possam acessar determinadas páginas. Essa separação de rotas públicas e rotas privadas garante a segurança da aplicação, impedindo que usuários sem um token JWT válido acessem rotas protegidas.

As rotas foram classificadas como públicas ou privadas. As rotas privadas só podem ser acessadas se o usuário estiver autenticado, ou seja, se o JWT armazenado for válido. Se o usuário não estiver autenticado, ele será redirecionado para a página de login. Segue o exemplo do código que realiza a validação:

```
1 const Private = ({ Item }) => {
2   const { isAuthenticated } = useContext(AuthContext);
3   return isAuthenticated() ? < Item /> : < Login />;
4   };
```

A função Private é responsável por proteger as rotas privadas. Ela utiliza o useContext para acessar o contexto de autenticação da aplicação, verificando se o usuário está autenticado através da função isAuthenticated() do AuthContext. Se o usuário estiver

autenticado, o componente protegido (Item) é renderizado. Caso contrário, o usuário é redirecionado para a página de login. Segue o exemplo do trecho de código que reliza a validação:

```
1 const RedirectIfAuthenticated = ({ Item }) => {
2   const { isAuthenticated } = useContext(AuthContext);
3   return isAuthenticated() ? < Item /> : < Login />;
4 };
```

A função RedirectIfAuthenticated trabalha de maneira similar, mas é utilizada para impedir que usuários autenticados acessem rotas que são destinadas a usuários não logados (por exemplo, a página de login). Se o usuário já estiver autenticado, ele é redirecionado para a página principal ou outro componente apropriado.

A configuração das rotas é feita no componente RoutesApp. Aqui, você pode definir quais rotas são públicas e quais são privadas. Para isso, usamos as funções Private e RedirectIfAuthenticated para proteger as rotas adequadamente.

```
1
2 const RoutesApp = () => {
3   return (
4     < Routes>
5       < Route path="/login" element={< RedirectIfAuthenticated Item={Home}
            />} />
6       < Route path="/imovel/cadastro" element={< Private Item={ImovelForm}
            />} />
7       < Route path="/prestador-servico/cadastro" element={< Private Item={
            PrestadorServicoForm} />} />
8       < Route path="/locatario/cadastro" element={< Private Item={
            LocatarioForm} />} />
9       < Route path="/imovel/busca" element={} />
10      < Route path="/imobiliaria/cadastro" element={< Private Item={
            ImobiliariaForm} />} />
11      < Route path="/relatorio/despesa/apartir-ano" element={< Private Item={
            RelatorioAnual} />} />
12      < Route path="/relatorio/aluguel/por-periodo" element={< Private Item={
            RelatorioAnualAlugueis} />} />
13      < Route path="/home" element={< Home />} />
14      < Route path="*" element={< Home />} />
15    < /Routes>
16  );
17 };
```

8.4.5 Detalhamento das Rotas

Rotas Públicas: A rota /login utiliza o componente RedirectIfAuthenticated. Se o usuário já estiver autenticado, ele será redirecionado para a página Home. Caso contrário, o usuário poderá acessar a rota de login.

Rotas Privadas: Rotas como /imovel/cadastro, /prestador-servico/cadastro, e outras semelhantes são protegidas pelo componente Private. Essas rotas só podem ser acessadas por usuários autenticados. Se o token JWT for válido, o usuário pode acessar a página desejada. Caso contrário, será redirecionado para a página de login.

Rota Padrão: A rota asterisco “*” é a rota curinga, que direciona o usuário para a página Home se ele tentar acessar uma rota inexistente.

Com essa configuração, a aplicação possui uma estrutura de rotas protegidas, onde:

As rotas públicas são acessíveis a qualquer usuário, independentemente de estar autenticado ou não, um exemplo é a rota Home e a de Login.

As rotas privadas só podem ser acessadas por usuários autenticados. Caso contrário, o usuário será redirecionado para a página de login.

Isso é feito de maneira eficiente usando o React Router em conjunto com o React Context para verificar a autenticação. A função isAuthenticated() dentro do contexto verifica se o token JWT é válido e, com base nisso, permite ou restringe o acesso às rotas.

Além disso, a API está preparada para receber diferentes perfis de usuário, como administrador ou usuário comum. Porém, como o sistema tende inicialmente a ser utilizado apenas por administradores, não foi necessário controlar o perfil do usuário que pode ou não ter acesso a uma rota especifica no frontend. No trecho de código abaixo podemos ver como o filtro é utilizado para verificar se é um usuário do tipo ADMINISTRADOR:

```
1         @PostMapping
2         @PreAuthorize("hasRole('ADMINISTRADOR')")
3         @Operation(summary = "Cadastra um novo Aluguel", description = "Acess
              ível somente para ADMINISTRADOR")
4         public ResponseEntity< AluguelResponse > salvar(@RequestBody
              @Validated AluguelRequest aluguelRequest) {
5
6                 return ResponseEntity.status(HttpStatus.CREATED).body(service
                      .salvar(aluguelRequest));
7         }
```

A anotação @PreAuthorize(“hasRole(’ADMINISTRADOR’)”) é a principal responsável por restringir o acesso ao método. Essa anotação, fornecida pelo Spring Security, permite definir uma expressão que deve ser verdadeira para que o método seja executado.

- *hasRole(’ADMINISTRADOR’)*: Esta expressão indica que o usuário que está tentando acessar o endpoint deve possuir o perfil ADMINISTRADOR. Se o usuário não tiver esse perfil, ele não poderá acessar esse recurso, e o Spring Security retornará uma resposta de erro, como 403 Forbidden.

8.4.6 Configuração de Segurança

Para garantir a segurança da aplicação, o Spring Security foi configurado para proteger endpoints de maneira flexível. A configuração foi feita através de componentes e anotada para facilitar a manutenção e escalabilidade. Abaixo está a configuração principal de segurança, utilizando a classe *ConfiguracaoSeguranca*:

```
1 @Configuration
```

```
@EnableWebSecurity
@RequiredArgsConstructor
@EnableMethodSecurity
public class ConfiguracaoSeguranca {

    private static final String[] ENDPOINTS_COM_PERMISSAO_TOTAL = {
          "/swagger-ui/**",
          "/v3/api-docs/**",
          "/actuator/health",
          "/v1/api/auth/login"
    };
    private final OncePerRequestFilter oncePerRequestFilter;
    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws
        Exception {
        return http
                .csrf(AbstractHttpConfigurer::disable)
                .sessionManagement(session -> session.sessionCreationPolicy(
                    SessionCreationPolicy.STATELESS))
                .authorizeHttpRequests(authorize ->
                        authorize
                                .requestMatchers(ENDPOINTS_COM_PERMISSAO_
                                    TOTAL).permitAll()
                                .anyRequest().authenticated()
                )
                .addFilterBefore(oncePerRequestFilter,
                    UsernamePasswordAuthenticationFilter.class)
                .build();
    }
    @Bean
    public AuthenticationManager authenticationManager(
        AuthenticationConfiguration authenticationConfiguration) throws
        Exception {
        return authenticationConfiguration.getAuthenticationManager();
    }
    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }
    @Bean
    public CorsFilter corsFilter() {
        UrlBasedCorsConfigurationSource source = new
            UrlBasedCorsConfigurationSource();
        CorsConfiguration config = new CorsConfiguration();
        config.addAllowedOrigin("*");
        config.addAllowedMethod("*");
        config.addAllowedHeader("*");
        source.registerCorsConfiguration("/**", config);
        return new CorsFilter(source);
    }
}
```

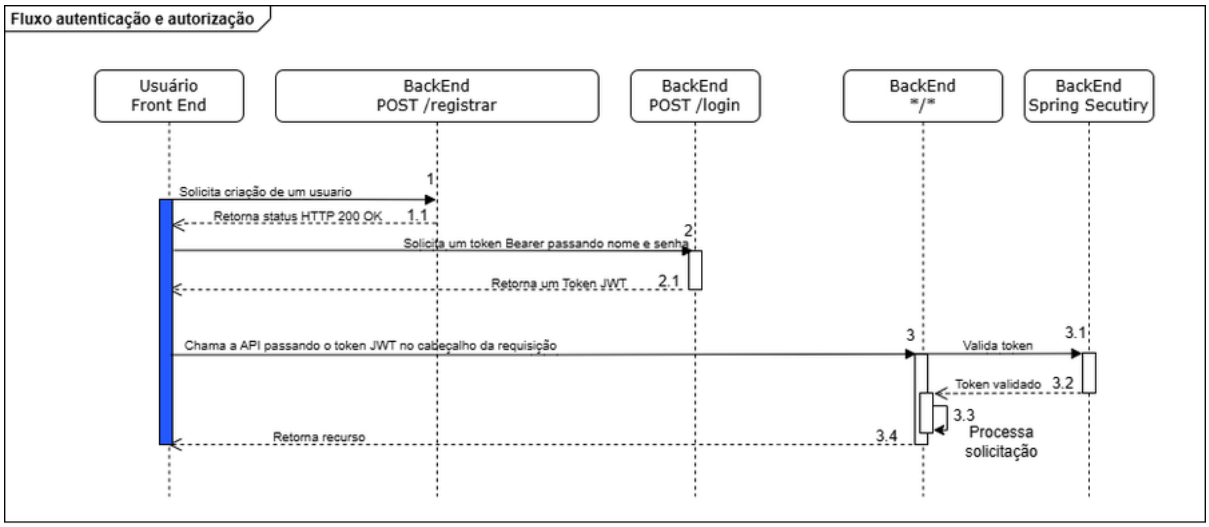
Este filtro garante que somente usuários autenticados, com tokens válidos, possam acessar os recursos protegidos da aplicação.

Repare que o método corsFilter() cria um filtro CorsFilter} que filtra todas as origens, métodos e cabeçalhos em requisições cross-origin e é configurado para permitir qualquer origem, qualquer método HTTP (como GET, POST, etc.), e qualquer cabeçalho HTTP. É uma configuração permissiva, adequada para desenvolvimento e ambientes onde uma política de CORS mais restritiva não é necessária. Em ambientes de produção, recomenda-se ajustar as permissões de CORS para atender às necessidades de segurança específicas de cada projeto.

A variável *ENDPOINTS_COM_PERMISSAO_TOTAL é utilizada para liberar acesso aos endpoint que não necessitão de autenticação para serem acessados, tornando a manutenção fácil caso seja necessário restringir ou não um endpoint.*

Na Figura 10, é possível visualizar o diagrama do fluxo de autenticação e autorização da aplicação. Inicialmente, é necessário criar um usuário no sistema, enviando uma requisição ao endpoint de registro (Passo 1). Em seguida, o usuário pode autenticar-se no sistema através do endpoint de login, onde será gerado um token JWT (Passo 2). Com o usuário autenticado e o token gerado, é possível realizar requisições aos endpoints da API (Passo 3). O token, então, é validado pelo Spring Security (Passo 3.1). Caso o token seja válido e o usuário possua as permissões necessárias para acessar o recurso, a aplicação dará prosseguimento ao processamento da solicitação (Passo 3.3) e retornará o recurso solicitado ao front-end (Passo 3.4).

Figura 10 – Diagrama de Sequência da Camada de Autenticação e Autorização



Fonte: autoria própria (2024)

Assim, o fluxo de autenticação e autorização garante a segurança da aplicação, assegurando que apenas usuários autenticados e autorizados possam acessar recursos específicos, mantendo a integridade dos dados e a proteção contra acessos não permitidos.

8.5 IMPLEMENTAÇÃO DO CI/CD

Neste capítulo, será abordada a implementação prática de CI/CD no projeto, utilizando GitHub Actions para automatizar a pipeline de build e deploy. A combinação dessas ferramentas permitiu integrar de forma contínua o código, realizar testes automatizados, construir imagens Docker e enviá-las ao Docker Hub, garantindo a entrega contínua do software de maneira eficiente e segura.

Para implementar o pipeline de CI/CD, foram utilizadas diversas ferramentas e sistemas, onde se destaco o uso do Github Actions, Docker, Docker Hub, Heroku, Gradle e Jacoco.

8.5.1 GitHub Actions

O GitHub Actions é um serviço de automação oferecido pelo GitHub para ajudar a automatizar o ciclo de desenvolvimento de software. Com ele, você pode configurar pipelines para etapas como build (compilação), testes e deploy (implantação). Esses pipelines são definidos por meio de arquivos YAML no repositório GitHub, e cada workflow (fluxo de trabalho) é acionado por eventos, como o push de código, criação de pull requests, ou a criação de uma nova tag. Em um pipeline CI/CD, o GitHub Actions permite garantir que cada alteração no código seja automaticamente testada e validada antes de ser integrada ou publicada.

8.5.2 Docker

Docker é uma plataforma de conteinerização que cria contêineres, que são ambientes isolados e portáteis, garantindo que o software funcione da mesma forma em diferentes máquinas.

No contexto do pipeline, o Docker é usado para construir uma imagem do projeto, incluindo o código e suas dependências, garante que o mesmo ambiente seja replicado em produção e desenvolvimento, minimizando problemas de inconsistência entre esses ambientes.

Desta forma ele facilita o deploy do aplicativo, pois a imagem criada pode ser facilmente transportada e executada em qualquer servidor que suporte Docker.

8.5.3 Docker Hub

O Docker Hub é um repositório público para armazenar e compartilhar imagens Docker. Em um pipeline CI/CD, como o configurado com o GitHub Actions, a cada nova release ou criação de tag, uma nova imagem Docker da API é criada e enviada ao Docker Hub, permitindo que a versão mais recente do aplicativo esteja disponível e pronta para deploy.

8.5.4 Gradle

O Gradle é uma ferramenta de automação de builds altamente personalizável, projetada para gerenciar dependências, compilar código e facilitar a criação de aplicativos em várias plataformas. Ele é amplamente usado para projetos Java, Kotlin, e Android,

permitindo a execução incremental e paralela de tarefas para maximizar a eficiência. Além disso, Gradle oferece um rico ecossistema de plugins e é compatível com diversas IDEs, como IntelliJ IDEA e Eclipse, além de ter integração com sistemas de CI. (GRADLE, 2024a)

No contexto do pipeline CI/CD, o Gradle foi utilizado para compilar o código-fonte, preparando-o para ser empacotado em uma imagem Docker, executa testes, verificando a qualidade e a funcionalidade do código e gera relatórios de cobertura de testes com o Jacoco, assegurando que o requisito de cobertura seja atingido antes de prosseguir com o deploy.

8.5.5 Jacoco

O JaCoCo é um plugin para Gradle que oferece métricas de cobertura de código Java, permitindo visualizar quais partes do código são testadas. Ele gera relatórios em formatos como HTML e XML e permite a verificação automática de metas de cobertura, onde o build falha caso as metas não sejam atendidas. A configuração do JaCoCo é flexível, e ele pode ser integrado com outras ferramentas de testes e relatórios no Gradle.

8.5.6 Heroku

É uma plataforma como serviço (PaaS) que permite construir, implementar e gerenciar aplicações na nuvem com facilidade. Ele oferece suporte a várias linguagens de programação e simplifica o processo de deploy ao lidar com a infraestrutura, provisionamento de servidores e configuração, permitindo que os desenvolvedores se concentrem em escrever código. Além disso, o Heroku facilita a integração contínua (CI/CD) e fornece ferramentas para escalar aplicações, gerenciar dados e monitorar performance (Heroku, 2024).

8.5.7 Configuração do Github Actions

A pipeline de Integração Contínua foi configurada para garantir que, a cada alteração nas branches main ou develop, o código fosse validado através de uma série de etapas automatizadas. Além disso, quando uma nova release era criada no GitHub, a pipeline também faz o build e deploy da aplicação.

O arquivo de configuração do workflow YAML responsável pelo CI/CD é o gradle.yaml, e deve ser salvo na pasta “.github/workflow/”, assim o GitHub Actions irá encontrar as configurações e executará os passos configurados no arquivo. A seguir será demonstrado a configuração básica do arquivo gradle.yaml, responsável pelo CI/CD:

```
1 name: Java CI com Gradle e Docker
```

```
 2
 3 on:
 4   push:
 5     branches:
 6       - main
 7       - develop
 8   pull_request:
 9     branches:
10       - main
11       - develop
12   release:
13     types: [created] # Aciona o deploy quando uma nova release é criada
14
15 jobs:
16   build:
17
18     runs-on: ubuntu-latest
19     permissions:
20       contents: read
21
22     steps:
23       - uses: actions/checkout@v4
24       - name: Set execute permission on gradlew
25         run: chmod +x gradlew
26       - name: Set up JDK 17
27         uses: actions/setup-java@v4
28         with:
29           java-version: '17'
30           distribution: 'temurin'
31       - name: Setup Gradle
32         uses: gradle/actions/setup-
               gradle@417ae3ccd767c252f5661f1ace9f835f9654f2b5 # v3.1.0
33
34       - name: Build
35         run: ./gradlew clean build
36       - name: Teste e Cobertura
37         run: ./gradlew test jacocoTestReport jacocoTestCoverageVerification
```

Nesta etapa, foi configurado um pipeline para rodar em eventos de push e pull request nas branches main e develop, além de releases criadas no repositório. O Java foi configurado com a versão 17, usando a distribuição Temurin, e o Gradle foi preparado para o processo de build e testes.

A cobertura de testes é validada com o Jacoco, executando o comando na linha 37, garantindo que a API tenha, no mínimo, 80% de cobertura de testes unitários antes que o código seja aprovado para deploy.

8.5.8 Entrega Continua com Docker

A etapa de deploy automatizado foi implementada utilizando Docker para criar e enviar imagens da aplicação ao Docker Hub[3]. O processo foi configurado para ser acionado sempre que uma nova release fosse criada no GitHub, garantindo que a imagem mais recente da aplicação fosse disponibilizada de forma automática.

A publicação para o docker foi configurado com a ajuda do actions checkout@v4, que contém uma pré-configuração feita pelo GitaActions para ajudar no processo de login e uso do CLI do docker como demonstrado no trecho abaixo:

[3] Disponível em: https://hub.docker.com/. Acesso em: 24 nov. 2024.

```
1   deploy:
2
3     needs: build
4
5     runs-on: ubuntu-latest
6     permissions:
7       contents: read
8
9     if: github.event_name == 'release'
10
11    steps:
12      - uses: actions/checkout@v4
13      - name: Set execute permission on gradlew
14        run: chmod +x gradlew
15      - name: Set up JDK 17
16        uses: actions/setup-java@v4
17        with:
18          java-version: '17'
19          distribution: 'temurin'
20
21      - name: Setup Gradle
22        uses: gradle/actions/setup-
              gradle@417ae3ccd767c252f5661f1ace9f835f9654f2b5 # v3.1.0
23
24      - name: Build
25        run: ./gradlew clean build
26
27      - name: Login Docker Hub
28        run: docker login -u ${{secrets.DOCKER_USERNAME}} -p ${{secrets.
              DOCKER_PASSWORD}}
29
30      - name: Build docker image
31        run: docker build --build-arg DB_URL= --build-arg DB_USERNAME= --
              build-arg DB_PASSWORD= --build-arg JWT_SECRET= -t usuario/larcerto
              -api:${{ github.event.release.tag_name }} .
32
33      - name: Push image docker
34        run: docker push usuario/larcerto-api:${{ github.event.release.tag_
              name }}
```

Nesta parte do pipeline o login no Docker Hub é realizado automaticamente utilizando as credenciais configuradas nos GitHub Secrets, então a imagem Docker é construída com base no código e nos argumentos necessários (como URL do banco de dados e segredo do JWT) e finalmente, a imagem da API é enviada ao Docker Hub, utilizando a tag da release criada no GitHub.

Essa combinação de ferramentas permite um pipeline de CI/CD robusto, que verifica automaticamente a integridade e a qualidade do código a cada atualização, construindo e enviando novas versões para o Docker Hub, prontas para serem implantadas.

8.5.9 Automação do Deploy no Heroku

A etapa de deploy também foi automatizada dentro do fluxo de CI/CD, onde foi utilizado a plataforma do Heroku para garantir a automação do deploy automatizado da aplicação, foi utilizada uma configuração no arquivo gradle.yml do GitHub Actions que compreende as seguintes etapas principais:

Login no Heroku Container Registry: Para autenticar com o Container Registry do Heroku, a chave secreta HEROKU_API_KEY foi configurada no github para conter o token de

acesso ao heroku, ela é acessada e usada para efetuar o login via Docker CLI. Isso permite a autenticação sem a necessidade de inserção manual da senha, garantindo segurança. Para esta configuração foi necessário inserir a configuração a seguir, responsável por realizar o login no Heroku:

```
1 - name: Login to Heroku Container Registry
2   env:
3     HEROKU_API_KEY: ${{ secrets.HEROKU_API_KEY }}
4   run: |
5     echo "$HEROKU_API_KEY" | docker login --username=_ --password-stdin
          registry.heroku.com
```

Tag da Imagem Docker para o Heroku: Após a criação da imagem Docker, é necessário associar a tag correspondente ao repositório de imagens do Heroku. Isso é feito utilizando o tagueamento adequado da imagem gerada com base no nome da aplicação configurada no Heroku:

```
1 - name: Tag Docker Image for Heroku
2   run: docker tag andersonfuhr/larcerto-api:${{ github.event.release.tag_name
        }} registry.heroku.com/${{ secrets.HEROKU_APP_NAME }}/web
```

Push da Imagem Docker para o Heroku: Esta etapa faz o envio da imagem Docker devidamente tagueada para o registro de contêineres do Heroku, utilizando o comando docker push para disponibilizá-la na plataforma de destino:

```
1 - name: Push Docker Image to Heroku
2   run: docker push registry.heroku.com/${{ secrets.HEROKU_APP_NAME }}/web
```

Release no Heroku: Por fim, a aplicação é liberada no Heroku através do comando heroku container:release, que sinaliza a execução e publicação da nova versão da aplicação:

```
1 - name: Release on Heroku
2   env:
3     HEROKU_API_KEY: ${{ secrets.HEROKU_API_KEY }}
4   run: |
5     heroku container:release web --app ${{ secrets.HEROKU_APP_NAME }}
```

Esse processo finaliza a automação do ciclo de CI/CD automatizado. Desta forma é possível garantir com que a aplicação seja continuamente integrada e lançada, reduzindo intervenções manuais, agilizando o ciclo de desenvolvimento, e fornecendo consistência nas entregas com o uso do CI/CD.

8.5.10 Validação e cobertura de Testes Utilizando Jacoco

No pipeline CI/CD, o Jacoco foi utilizado para garantir um nível mínimo de cobertura de testes, como 80%, antes que uma nova versão seja lançada. Dessa forma, assegura-se que a maioria do código foi testada, reduzindo riscos de bugs e falhas. Caso essa métrica não seja atingida, o pipeline falha, impedindo o deploy da aplicação. Isso garante que o código mantido no repositório principal esteja sempre bem testado, reduzindo o risco de bugs em produção.

Através de configuração do JaCoCo no build.gradle foi feita a geração de relatórios de cobertura e regras para validação como descrito abaixo:

```
tasks.test {
    finalizedBy(tasks.jacocoTestReport)
}
tasks.jacocoTestReport {
    dependsOn(tasks.test)
}
tasks.jacocoTestReport {
    reports {
        xml.required = true
        csv.required = false
        html.outputLocation = layout.buildDirectory.dir("jacocoHtml")
    }
}

tasks.jacocoTestCoverageVerification {
    violationRules {
        rule {
            limit {
                minimum = "0.8".toBigDecimal()
            }
        }
        rule {
            isEnabled = false
            element = "CLASS"
            includes = listOf("org.gradle.*")
            limit {
                counter = "LINE"
                value = "TOTALCOUNT"
                maximum = "0.8".toBigDecimal()
            }
        }
    }
}
tasks.check {
    dependsOn("jacocoTestCoverageVerification")
}
```

Esta configuração do JaCoCo no build.gradle define a geração de relatórios de cobertura e regras para validação.

- *tasks.test* finaliza chamando jacocoTestReport, que cria relatórios XML e HTML.
- *jacocoTestCoverageVerification* aplica regras, exigindo um mínimo de 80% de cobertura.
    - *violationRules*: Agrupa todas as regras de verificação de cobertura.

    - *rule*: Define uma regra específica para cobertura.
        - limit: Especifica a meta mínima ou máxima de cobertura.
        - counter e value: Definem o tipo de métrica a verificar, como “LINE” ou “INSTRUCTION”.
    - element: Define o nível de cobertura, como “CLASS” ou “METHOD”.

- *tasks.check:* depende da verificação de cobertura, falhando se as regras não forem cumpridas.

O atributo isEnabled = false indica que essa regra específica não será aplicada durante a verificação de cobertura.

Uma segunda *rule*, focada em classes org.gradle.*, está desativada (isEnabled = false) e define um limite máximo de 80% para cobertura de linhas. Mesmo que a regra defina uma cobertura máxima de 80% para linhas (LINE) na contagem total (TOTALCOUNT) e limite a verificação a classes no pacote org.gradle.*, ela não afetará o processo de build, pois isEnabled está configurado para false.

Essas etapas garantem relatórios e conformidade com cobertura mínima, porém a etapa só será executada executando o comando abaixo no prompt:

```
1 ./gradlew test jacocoTestReport jacocoTestCoverageVerification
```

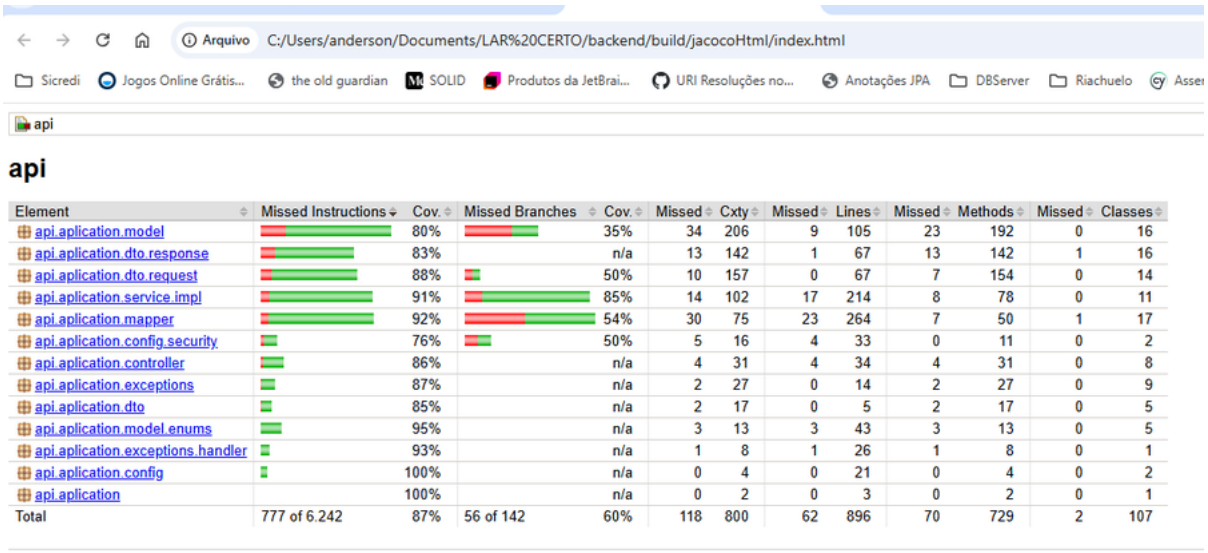
Este comando inclusive é executando na etapa de pipeline e realiza três ações importantes:

test: Executa os testes unitários do projeto, gerando dados de cobertura.

jacocoTestReport: Gera o relatório de cobertura de testes usando os dados coletados. Ele cria uma estrutura de pastas onde a pasta ./build/jacocoHtml/index.html contém o relatório HTML, que pode ser visualizado na figura 11.

jacocoTestCoverageVerification: Avalia a cobertura de testes com base nos critérios configurados no build, como a porcentagem mínima de linhas, branches, e classes cobertas. Se algum critério não for atendido, o Gradle lança um erro.

Figura 11 – Relatório Jacoco



Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
api.aplication.model		80%		35%	34	206	9	105	23	192	0	16
api.aplication.dto.response		83%		n/a	13	142	1	67	13	142	1	16
api.aplication.dto.request		88%		50%	10	157	0	67	7	154	0	14
api.aplication.service.impl		91%		85%	14	102	17	214	8	78	0	11
api.aplication.mapper		92%		54%	30	75	23	264	7	50	1	17
api.aplication.config.security		76%		50%	5	16	4	33	0	11	0	2
api.aplication.controller		86%		n/a	4	31	4	34	4	31	0	8
api.aplication.exceptions		87%		n/a	2	27	0	14	2	27	0	9
api.aplication.dto		85%		n/a	2	17	0	5	2	17	0	5
api.aplication.model.enums		95%		n/a	3	13	3	43	3	13	0	5
api.aplication.exceptions.handler		93%		n/a	1	8	1	26	1	8	0	1
api.aplication.config		100%		n/a	0	4	0	21	0	4	0	2
api.aplication		100%		n/a	0	2	0	3	0	2	0	1
Total	777 of 6.242	87%	56 of 142	60%	118	800	62	896	70	729	2	107

Fonte: autoria própria (2024)

A implementação de CI/CD com GitHub Actions, Docker e Jacoco permitiu automatizar de forma eficaz o ciclo de desenvolvimento, desde a validação do código com testes até a entrega contínua da aplicação em um ambiente de produção. Essa abordagem reduziu a possibilidade de erros e garantiu que novas versões da API pudessem ser disponibilizadas rapidamente com um processo de deploy seguro e confiável. Com a integração de ferramentas como Docker e Jacoco, foi possível garantir tanto a consistência no ambiente quanto a qualidade do código entregue.

8.6 VERSIONAMENTO DO BANCO DE DADOS

Neste projeto o Flyway automatiza as migrações do banco de dados em ambientes de teste e desenvolvimento, garantindo que a configuração do banco seja replicada fielmente. Isso possibilita a criação de testes end-to-end consistentes e realistas, nos quais qualquer mudança no esquema ou mapeamento das entidades é fielmente reproduzida, evitando divergências entre ambientes. Esse processo contribui para práticas de DevOps e CI/CD, permitindo uma evolução controlada do banco de dados ao longo do ciclo de vida do software.

Utilizei o PostgreSQL como banco de dados relacional, uma escolha alinhada com as necessidades de licença aberta e compatibilidade do Flyway.

Para garantir a consistência e o controle de versões do banco de dados no projeto em Java com Spring Boot, foi implementado o Flyway seguindo alguns passos detalhados a seguir:

Para se utilizar o Fyway no projeto é necesário importa-lo como uma dependência no arquivo *build.gradle.kts*. Abaixo segue a dependência a ser inserida:

```
1 dependencies {
2     implementation 'org.flywaydb:flyway-core'
3 }
```

Isso permite que o Gradle gerencie o Flyway e o integre ao projeto para que as migrações sejam aplicadas automaticamente.

Para conectar o Flyway ao banco de dados, foram definidas as configurações necessárias no arquivo application.properties do Spring Boot, como as credenciais e a URL de conexão ao banco. Também foi configurado o Flyway para que ele possa iniciar o banco a partir de uma baseline, definindo uma versão inicial. A configuração segue no código abaixo:

```
1 spring.flyway.url=${DB_URL}
2 flyway.baseline-on-migrate=true
3 spring.flyway.baseline-version=1
4 spring.flyway.user=${DB_USERNAME}
5 spring.flyway.password=${DB_PASSWORD}
6 flyway.locations=classpath:/db/migration
```

Para evitar que o JPA interferisse na criação das tabelas e que o Flyway pudesse controlar as alterações de esquema sem conflito é necessário desabilitar o gerenciamento do esquema pelo Hibernate com as seguintes opções no application.properties:

```
1 spring.jpa.hibernate.ddl-auto=none
2 spring.jpa.generate-ddl=false
```

Em seguida, criei o diretório onde os arquivos de migração do banco de dados ficariam armazenados, que o Flyway lê para aplicar as mudanças no banco. A pasta "db/migration" foi adicionada dentro de src/main/resources/.

Para nomear os arquivos de migração, deve se seguir a convenção exigida pelo Flyway, que facilita a organização e o controle sequencial das alterações. Por exemplo, para a primeira migração que cria a estrutura inicial do banco, o primeiro arquivo de migração foi nomeado como:

```
1 V01__init_database.sql
```

Esse padrão é importante, pois o Flyway reconhece cada parte da nomenclatura para controlar a ordem das migrações:

- "V" maiúsculo: Indica o início de uma nova migração.

- Versão: Identifica a sequência da migração (como 1, 1.1, etc.).
- Dois sublinhados (__): Delimitador obrigatório.
- Descrição: Ajuda a identificar o objetivo da migração.
- Extensão .sql: Identifica o arquivo como um script SQL.

Foi essencial garantir que a estrutura do banco de dados usada nos testes end-to-end fosse uma réplica fiel do ambiente de desenvolvimento. Para isso, configurei o Flyway para aplicar as migrações automaticamente na camada de testes, mantendo a consistência entre os ambientes. Essa configuração oferece uma base realista para os testes, permitindo que as interações e validações funcionem com a mesma estrutura de dados usada no desenvolvimento.

Para replicar o banco de dados no ambiente de testes, defini as mesmas configurações de conexão para os testes. Isso garantiu que o Flyway aplicasse todas as migrações na base de dados de testes, espelhando a estrutura do banco de desenvolvimento.

Para que os testes end-to-end fossem executados de forma leve e rápida, configurei o H2 como banco em memória no ambiente de testes, mockando a estrutura do banco principal. O H2 é um banco de dados relacional leve, e sua capacidade de rodar em memória facilita a criação e destruição rápida de dados a cada execução de teste. No arquivo application-test.properties, as configurações foram inseridas da seguinte forma:

```
1 #------------h2------
2 spring.datasource.url=jdbc:h2:mem:testdb;MODE=PostgreSQL;DB_CLOSE_DELAY=-1;DB
      _CLOSE_ON_EXIT=FALSE
3 spring.datasource.driver-class-name=org.h2.Driver
4 spring.datasource.username=sa
5 spring.datasource.password=
6 spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.H2Dialect
7 #-------------------------------------- HIBERNATE
      ----------------------------------------
8 spring.jpa.hibernate.ddl-auto=none
9 spring.jpa.generate-ddl=false
10 #-------------------------------------- FlyWay
      ----------------------------------------
11 spring.flyway.enabled=true
12 spring.flyway.locations=classpath:test-migration
```

Com o Flyway gerenciando as migrações no ambiente de testes, os testes end-to-end conseguem acessar uma estrutura de banco confiável e realista. Isso é especialmente importante para verificar o comportamento da aplicação em interações completas (como criação, leitura, atualização e exclusão de dados), validando que as funcionalidades da API se comportem corretamente em uma estrutura de banco autêntica.

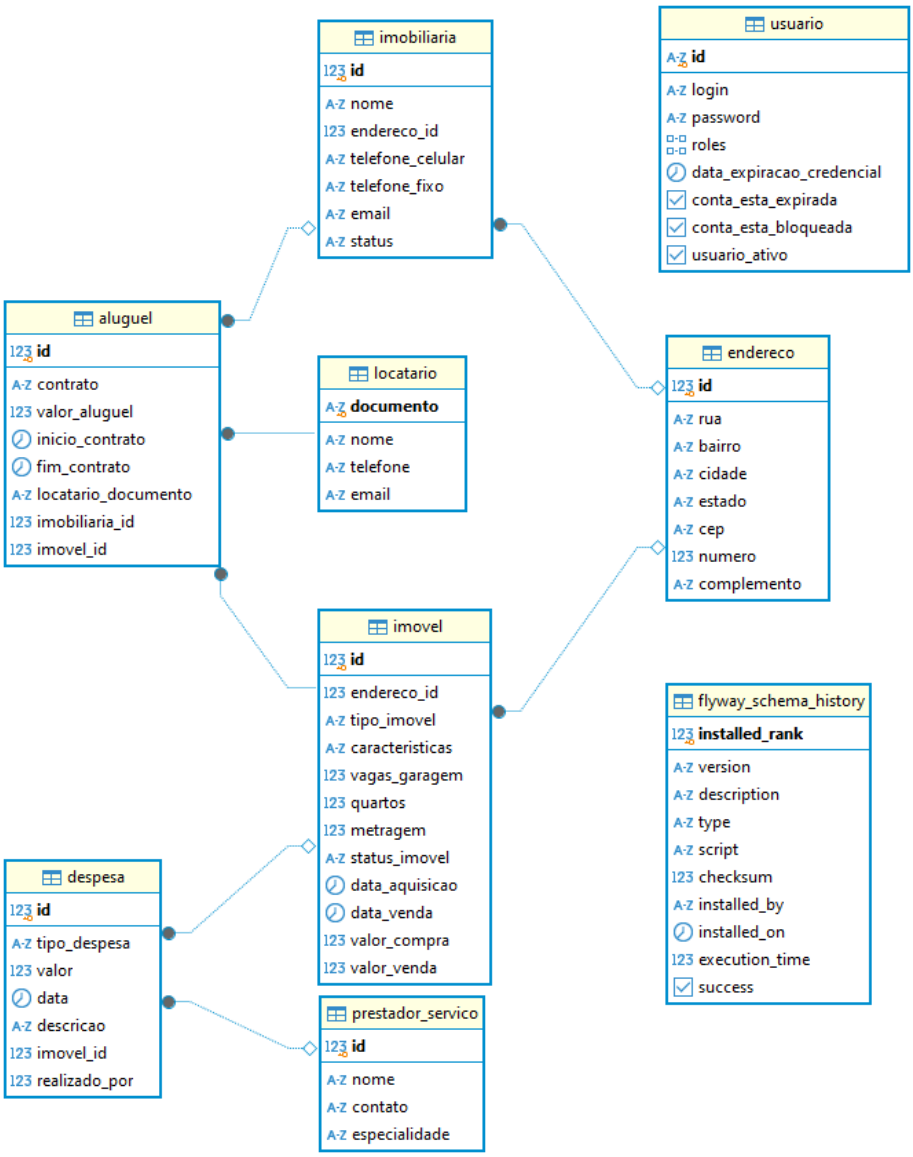
Essas configurações garantem que o Spring Boot use o H2 como banco de dados temporário durante os testes, em vez do banco principal (como o PostgreSQL), permitindo que o Flyway aplique as migrações automaticamente ao H2.

Esse ambiente controlado melhora a precisão dos testes, pois garante que quaisquer mudanças na estrutura ou mapeamento do banco sejam refletidas automaticamente nos testes, proporcionando uma experiência similar ao ambiente de produção. Como resultado, essa abordagem traz consistência ao ciclo de desenvolvimento e permite identificar e corrigir problemas antes da fase de implantação.

Essa configuração com o Flyway permitiu que eu mantivesse o banco de dados sincronizado entre diferentes ambientes e que as mudanças fossem aplicadas automaticamente durante o desenvolvimento e testes. Além de facilitar a integração com o processo de CI/CD, essa abordagem garante que o banco de dados evolua de maneira controlada e previsível ao longo do ciclo de vida do projeto.

O diagrama ER da figura 12 demonstra como o banco de dados esta mapeado:

Figura 12 – Schema do banco de dados da aplicação



Fonte: autoria própria (2024)

As tabelas demonstradas na figura 12 seguem o seguinte relacionamento e uso prático:

- *flyway_schema_history:* Tabela gerenciada automaticamente pelo Flyway para controlar as versões de migrações do banco de dados. Ela não participa diretamente na modelagem de domínio do sistema.
- *usuario:* Gerencia a autenticação e autorização de usuários do sistema e não tem relacionamentos com outras tabelas de domínio. Controla o acesso ao sistema, definindo permissões e armazenando credenciais com segurança.
- *endereco*: Representa os endereços associados a imóveis e imobiliárias, permitindo uma localização detalhada. Facilita a gestão de localizações para imóveis e imobiliárias, centralizando os dados geográficos em uma única tabela.
  - Cardinalidade e Relacionamentos:
    - 1:1 com imobiliaria: Uma imobiliária tem exatamente um endereço.
    - 1:1 com imovel: Um imóvel possui exatamente um endereço.
- *aluguel:* Armazena os contratos de aluguel, associando informações de locatários, imóveis e imobiliárias. Permite o registro e gestão de contratos de aluguel, incluindo valores, prazos e vínculos entre locatários e imóveis.
  - Cardinalidade e Relacionamentos:
    - N:1 com imobiliaria: Um aluguel pode ser gerido por uma imobiliária.
    - N:1 com imovel: Um aluguel está associado a um único imóvel.
    - N:1 com locatario: Um aluguel é realizado por um único locatário.
- *despesa*: Registra despesas relacionadas a imóveis e serviços prestados. Facilita o controle financeiro e histórico de gastos associados a imóveis, com detalhamento de quem realizou o serviço e qual o valor.
  - Cardinalidade e Relacionamentos:
    - N:1 com imovel: Uma despesa está vinculada a um único imóvel.
    - N:1 com prestador_servico: Uma despesa pode ser realizada por um único prestador de serviço.

- *imobiliaria:* Representa as imobiliárias responsáveis pela gestão de aluguéis e imóveis. Centraliza os dados de imobiliárias, associando imóveis e contratos a cada entidade gestora.

  - Cardinalidade e Relacionamentos:

    * 1:1 com endereco: Uma imobiliária está associada a um único endereço.
    * 1:N com aluguel: Uma imobiliária pode gerenciar vários contratos de aluguel.

- *imovel*: Representa os imóveis cadastrados no sistema, com informações como tipo, características e status. Tabela central para a gestão do sistema, permitindo cadastrar, gerenciar e rastrear o histórico de cada imóvel.

  - Cardinalidade e Relacionamentos:

    * 1:1 com endereco: Um imóvel está vinculado a um único endereço.
    * 1:N com aluguel: Um imóvel pode estar associado a vários aluguéis ao longo do tempo.
    * 1:N com despesa: Um imóvel pode ter várias despesas associadas.

- *locatario*: Representa as pessoas físicas ou jurídicas que alugam imóveis. Facilita o gerenciamento de informações de locatários, como dados de contato.

  - Cardinalidade e Relacionamentos:

    * 1:N com aluguel: Um locatário pode estar associado a vários contratos de aluguel ao longo do tempo.

- *prestador_servico:* Representa os profissionais ou empresas que realizam serviços para os imóveis. Permite a gestão de prestadores de serviço, vinculando seus dados às despesas registradas no sistema.

  - Cardinalidade e Relacionamentos:

    * 1:N com despesa: Um prestador de serviço pode ser associado a várias despesas realizadas.

8.7 QUALIDADE DE SOFTWARE

Cada funcionalidade desenvolvida no sistema seguiu o mesmo fluxo: desenvolvimento no backend, implementação de testes unitários no backend, implementação de testes E2E no backend, desenvolvimento no frontend e, por último, testes funcionais. Esse processo, que demandou um tempo considerável.

A pirâmide de testes foi implementada na API para garantir a qualidade e a confiabilidade do sistema, englobando diversas camadas de verificação. Esta abordagem abrangeu testes unitários, testes de integração e testes de ponta a ponta (E2E), utilizando o framework Spring e banco de dados em memória H2.

Como já mencionado anteriormente, foi utilizado ferramentas como o jacoco para monitoramento e validação dos testes, verificando a cobertura e qualidade de alcance e profundidade dos testes, inclusive integrando a pipeline de CI/CD para garantir a qualidade do desenvolvimento do sistema como um todo.

Também foi feito uso de ferramentas de análise de qualidade como SonarLint para o back end e ESLint para a aplicação frontend.

O ESLint permite definir quais regras serão aplicadas ao código. Por exemplo, ele pode definir que aspas simples devem ser usadas para strings, que linhas não podem ultrapassar determinado número de caracteres ou que variáveis não utilizadas devem ser alertadas. Essas regras podem ser configuradas para diferentes níveis de severidade:

error: indica que o ESLint deve considerar o problema como um erro que precisa ser corrigido.

warn: indica uma advertência, mas o código ainda poderá ser executado.

off: desativa a regra.

Para instalar e utilizar no frontend o Eslint precisa ser instalado como uma dependência no projeto. Para isso, deve se abrir um terminal de comando, navegar até a pasta raiz do projeto e executar o comando a seguir no terminal:

```
1 npm install eslint-plugin-react --save-dev
```

Após sua instalação no projeto React, deve ser adicionado a sua configuração no arquivo package.json onde as configurações básicas ajudaram a garantir uma qualidade mínima de codificação, onde as seguinte regras foram setadas:

semi: [“error”, “always”]: Exige o uso de ponto e vírgula no final de cada linha, emitindo um erro caso não esteja presente.

no-unused-vars: “error”: Marca como erro quaisquer variáveis declaradas, mas não utilizadas, ajudando a manter o código limpo e organizado.

no-console: “warn”: Emite um aviso ao usar console.log, incentivando a remoção de logs antes do deploy.

no-extra-semi: “error”: Impede o uso de ponto e vírgula extra, o que evita linhas desnecessárias.

no-multiple-empty-lines: [“error”, { “max”: 3 }]: Define um erro para linhas em branco consecutivas que excedam três, ajudando a manter o código visualmente organizado.

eqeqeq: “error”: Exige o uso de igualdade estrita (===) ao invés de igualdade simples (==), evitando erros de comparação entre tipos diferentes no uso de IF dentro do código.

O trecho abaixo demonstra a configuração do arquivo ./package.json:

```
"eslintConfig": {
  "extends": [
    "react-app",
    "react-app/jest"
  ],
  "rules": {
    "semi": [
      "error",
      "always"
    ],
    "no-unused-vars": "error",
    "no-console": "warn",
    "no-extra-semi": "error",
    "no-multiple-empty-lines": [
      "error",
      {
        "max": 3
      }
    ],
    "eqeqeq": "error"
  }
}
```

Desta forma diversas outras rules também poderiam ser adicionadas e mais informações podem ser encontradas diretamente no site que contém a documentação “*https ://eslint.org/docs/latest*“, onde poderia ser extendido e modificado as regras conforme a necessidade atual do projeto.

Uma observação importante é que o Eslint também foi integrado a execução da pipeline de CI/CD, ajudando que somente o código que não contenha erros pré-estabelecidos sigam para produção.

SonarLint é uma extensão leve para IDEs (incluindo IntelliJ) que permite fazer análise de código em tempo real enquanto você desenvolve. Ele destaca problemas de qualidade e más práticas de programação à medida que você escreve o código, verificando métricas de cobertura de testes e destacando pontos de refatoração necessários. Através dessa ferramenta, foi possível identificar problemas como duplicação de código, complexidade elevada e potenciais falhas de segurança, auxiliando na manutenção de um código alinhado com as melhores práticas.

Para instalação e uso no IntelliJ vá até o menu “*File* $>$ *Settings* $>$ *Plugins*“ e procure por ”sonarlint“, clique em instalar e reinicie a IDE que poderá ter uma ferramenta que irá analisar seu código emitindo alertas de erro e sugestões de melhoria. Mais detalhes podem ser encontrados na documentação oficial da ferramenta ”https://plugins.jetbrains.com/plugin

/7973-sonarlint“.

Junto a isso foi importante tentar garantir que o software se comporte de maneira mais estável em diversos navegadores, e para isso usei outra ferramenta chamada Babel. O Babel ajuda a escrever código JavaScript moderno sem se preocupar com problemas de compatibilidade em navegadores que ainda não suportam esses recursos. Isso permite que desenvolvedores usem os recursos mais recentes da linguagem, aumentando a produtividade e a legibilidade do código, enquanto garantem que o código final seja compatível com uma ampla gama de navegadores e dispositivos.

Usado principalmente para converter código ECMAScript 2015+ em uma versão compatível com versões anteriores do JavaScript em navegadores ou ambientes atuais e mais antigos, garante a estabilidade do sistema (BABEL, 2024).

Ele também é utilizado como dependência no projeto React e pode ser configurado no arquivo package.json, para isso é necessário executar este comando de instalação de dependência:

```
1 npm install @babel/preset-env --save-dev
```

Após a instalação da dependência, pode ser então configurado a lista de navegadores suportados, inserindo o trecho a de código a seguir no arquivo package.json:

```
1 "browserslist": {
2     "production": [
3       ">0.2%",
4       "not dead",
5       "not op_mini all"
6     ],
7     "development": [
8       "last 1 chrome version",
9       "last 1 firefox version",
10      "last 1 safari version"
11    ]
12  }
```

- production: Define os navegadores a serem suportados em produção.
  - *>0.2%*: Inclui navegadores que são usados por mais de 0,2% dos usuários globalmente.
  - *not dead*: Exclui navegadores que não recebem mais atualizações de segurança.
  - *not op_mini all*: Exclui todos os navegadores Opera Mini, que têm um suporte limitado a JavaScript.
- development: Define os navegadores que serão usados durante o desenvolvimento.

- *“last 1 chrome version”, “last 1 firefox version”, “last 1 safari version”*: Garante compatibilidade com a última versão dos navegadores Chrome, Firefox e Safari, o que ajuda a simplificar e otimizar o ambiente de desenvolvimento, garantindo que o projeto funcione nas versões mais recentes desses navegadores.

Com essa configuração no browserslist é explicitamente indicado que a aplicação não oferece suporte ao Opera Mini. O parâmetro “not op_mini all” exclui todas as versões do Opera Mini, conhecido por ter um suporte limitado a recursos modernos de JavaScript e CSS devido à sua natureza de compressão extrema e otimização para dados móveis.

No entanto, isso não significa que a aplicação não funcionará no navegador Opera padrão. O Opera é baseado no mesmo mecanismo de renderização do Chrome (Blink), então ele geralmente oferece suporte a recursos modernos de JavaScript e CSS semelhantes aos do Chrome. Portanto, apenas o Opera Mini é excluído da lista de navegadores suportados, enquanto o Opera padrão deve funcionar corretamente.

8.7.1 IMPLEMENTAÇÃO DE TESTES END-TO-END(E2E) NA API

A implementação dos testes E2E automatizados foi fundamental para verificar o comportamento da aplicação em cenários reais, simulando interações completas com a API. Para isso, recriei um ambiente de testes configurado como um ambiente de produção simplificado, utilizando um módulo de testes fornecido pelo Spring Boot e um banco de dados em memória do tipo H2.

O framework Spring Boot oferece um módulo específico para testes, permitindo o desenvolvimento de testes E2E com uma configuração simplificada e integrada. Esse módulo vem pré-configurado para realizar a inicialização automática dos componentes da aplicação, como o contexto do Spring, facilitando o acesso e a interação com os endpoints da API de forma realista. Com essa abordagem, foi possível inicializar uma instância da API diretamente no ambiente de testes, garantindo que os endpoints estivessem acessíveis para cada cenário de teste.

Para simular um ambiente de produção durante os testes, utilizei o banco de dados em memória H2, que permitiu a criação de um banco temporário, isolado e de rápida inicialização para cada execução de teste. A configuração do H2 no modo de teste proporcionou um ambiente de persistência temporária e semelhante ao banco de produção, permitindo que a API manipulasse dados sem depender de uma conexão real.

Para tornar os testes mais realistas e permitir que a aplicação interagisse com dados completos, criei scripts SQL que eram executados automaticamente no banco H2 ao iniciar cada teste. Esses scripts SQL populam o banco de dados em memória com dados mock, fornecendo um ambiente com registros pré-definidos que simulavam as condições de produção. Isso garantiu que cada teste tivesse um conjunto de dados consistente e

previsível, facilitando a verificação dos comportamentos e o acesso aos endpoints com diferentes cenários de dados.

Como a API utiliza Spring Security para controle de autenticação e autorização, foi necessário incluir a biblioteca Spring Security Test. Com essa biblioteca, simulei diferentes tipos de usuários e níveis de permissão nos testes, o que permitiu testar cenários variados de acesso a cada endpoint, garantindo que apenas usuários autorizados pudessem acessar determinadas funcionalidades e que o sistema respondesse adequadamente a tentativas de acesso não autorizadas.

Com o ambiente configurado e os dados carregados, a execução dos testes E2E permitiu verificar o comportamento da aplicação ao acessar diretamente os endpoints da API. Foram realizadas simulações de chamadas HTTP para avaliar a resposta da API em condições de uso semelhantes às de produção. Durante os testes, cada endpoint foi avaliado quanto à resposta esperada, como códigos de status HTTP, mensagens de erro apropriadas e integridade dos dados retornados.

A classe BaseIntegrationTest foi criada como uma classe base para os testes de integração, centralizando as configurações e os componentes essenciais para todos os testes. Essa abordagem simplifica e modulariza a configuração dos testes, tornando-a mais reutilizável e reduzindo redundâncias. A seguir é demonstrado a classe BaseIntegrationTest, que foi utilizada para configurar os demais testes:

```
1 @SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
2 @AutoConfigureMockMvc
3 @SqlGroup({
4       @Sql(executionPhase = Sql.ExecutionPhase.BEFORE_TEST_METHOD, scripts =
            MOCK_DATABASE),
5       @Sql(executionPhase = Sql.ExecutionPhase.AFTER_TEST_METHOD, scripts =
            RESETAR_DB)
6 })
7 public abstract class BaseIntegrationTest {
8
9     @Autowired
10    protected MockMvc restClient;
11
12    @Autowired
13    protected final ObjectMapper mapper = new ObjectMapper();
14
15    protected DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd-
          MM-yyyy");
16 }
17 }
```

A seguir está uma explicação de cada trecho e dos benefícios dessa técnica:

@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT): Inicializa o contexto da aplicação em um ambiente de teste, permitindo que o Spring Boot crie uma instância da aplicação na porta aleatória, simulando um ambiente real sem interferir em outras instâncias.

@AutoConfigureMockMvc: Configura o MockMvc, que permite simular e testar endpoints HTTP da aplicação sem a necessidade de subir um servidor web real.

@SqlGroup: Executa scripts SQL antes e depois de cada método de teste, garantindo que o banco de dados esteja em um estado conhecido e consistente. O script MOCK_DATABASE popula o banco com dados iniciais antes de cada teste, e o RESETAR_DB limpa o banco após a execução. Essa prática ajuda a manter os testes isolados e independentes entre si.

MockMvc (restClient): Permite simular requisições HTTP (como GET, POST, PUT, DELETE) diretamente aos endpoints da aplicação, validando as respostas sem precisar de um servidor real.

ObjectMapper (mapper): Usado para conversão de objetos Java para JSON e vice-versa, facilitando a manipulação de dados nos testes.

DateTimeFormatter (formatter): Formata as datas em um padrão específico (dd-MM-yyyy), padronizando as representações de data em todos os testes.

Ao criar essa classe base, foi modularizado a configuração de testes de forma que outras classes de teste possam simplesmente estender BaseIntegrationTest e herdar todas essas configurações automaticamente. Isso traz várias vantagens:

Reutilização de Código: As classes de teste herdam as configurações e dependências comuns, eliminando duplicação de código e mantendo consistência entre os testes.

Simplicidade e Clareza: Cada classe de teste individual se torna mais simples, já que não precisa configurar novamente componentes como MockMvc, ObjectMapper, e formatação de data. Isso também facilita o entendimento dos testes, uma vez que cada um se concentra no que está sendo testado e não na configuração.

Manutenção Facilitada: Se for necessário ajustar qualquer configuração, basta modificar a classe BaseIntegrationTest, e todas as classes de teste que a estendem serão atualizadas automaticamente. Isso torna o código de teste mais fácil de manter e escalável.

Ambiente de Teste Consistente: Com o uso de scripts SQL antes e depois de cada teste, o banco de dados é mantido em um estado conhecido, eliminando dependências entre os testes e assegurando que cada execução comece com os mesmos dados iniciais.

Essa técnica contribui para testes mais organizados, coesos e reutilizáveis, que seguem boas práticas de desenvolvimento e oferecem uma base modular para o crescimento do projeto.

Esse teste abaixo na classe LocatarioControllerTest utiliza a classe base BaseIntegrationTest para herdar configurações essenciais que simplificam a simulação de requisições e o acesso ao ambiente de teste.

```
1 class LocatarioControllerTest extends BaseIntegrationTest {
2
3         @Test
4         @WithMockUser(roles = {"ADMINISTRADOR"})
5         @DisplayName("Deve salvar um Locatario na base de dados corretamente")
6         void salvarLocatario() throws Exception {
7
8                 LocatarioRequest locatarioRequest = LocatarioRequestFixture.builder();
9                 String requisicao = mapper.writeValueAsString(locatarioRequest);
10
```

```
11                  this.restClient.perform(post("/v1/api/locatario/")
12                                                  .contentType(MediaType.APPLICATION_
                                                        JSON)
13                                                  .content(requisicao))
14                                  .andExpect(status().isCreated())
15                                  .andExpect(jsonPath("$.nome").value(locatarioRequest.
                                        getNome()))
16                                  .andExpect(jsonPath("$.documento").value(
                                        locatarioRequest.getDocumento()))
17                                  .andExpect(jsonPath("$.email").value(locatarioRequest.
                                        getEmail()))
18                                  .andExpect(jsonPath("$.telefone").value(
                                        locatarioRequest.getTelefone()));
19          }
```

Herança de MockMvc (restClient): Através do restClient herdado, o teste consegue simular uma requisição POST ao endpoint /v1/api/locatario/, criando um locatário sem precisar configurar o MockMvc na própria classe de teste.

Uso do ObjectMapper (mapper): Através do mapper herdado, o teste converte o objeto locatarioRequest em JSON, facilitando o envio dos dados no formato correto para a API.

Controle de Acesso com @WithMockUser: A anotação @WithMockUser(roles = {“ADMINISTRADOR”}) simula um usuário com o papel de administrador, necessário para que o teste valide a autorização e autenticação configuradas no Spring Security.

Esse teste verifica se o endpoint “/v1/api/locatario/” cria um locatário na base de dados corretamente, validando o status de resposta e os dados do locatário retornado, tudo com suporte das configurações herdadas da classe BaseIntegrationTest.

9 DESCRIÇÃO DAS FUNCIONALIDADES IMPLEMENTADAS

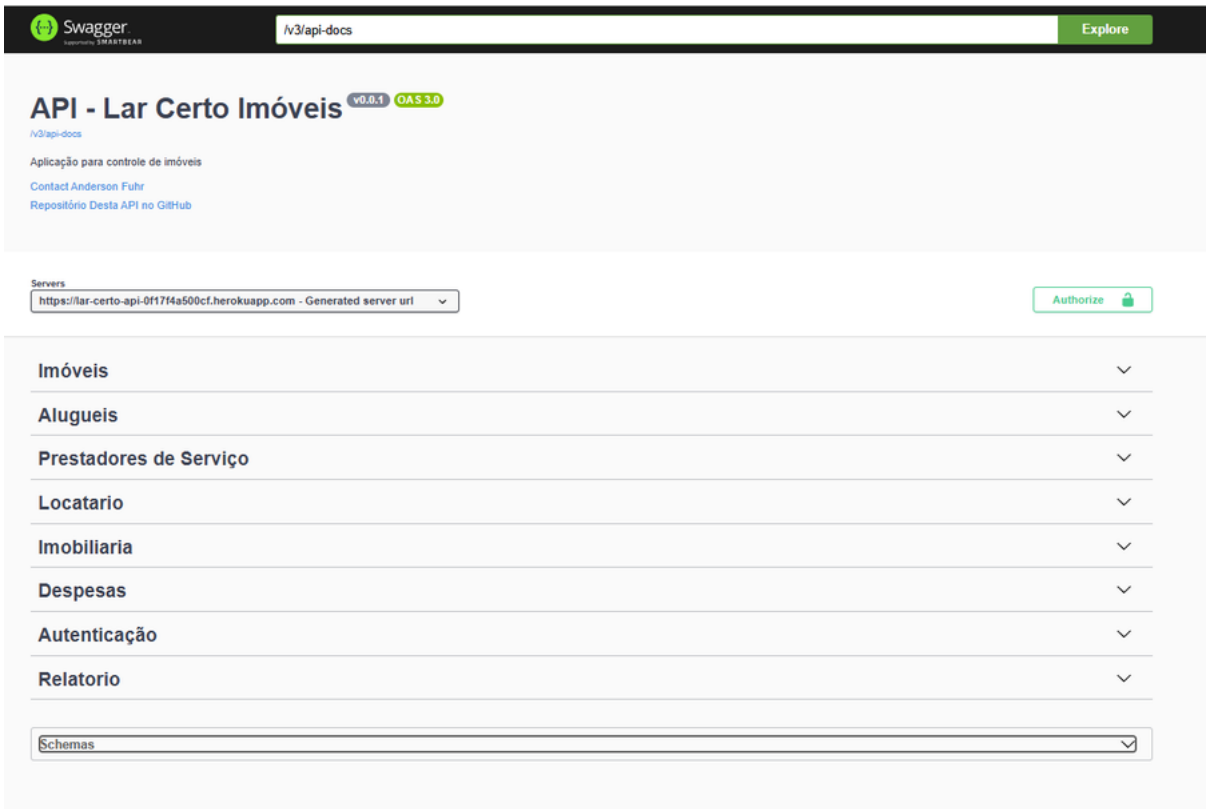
Este capítulo visa apresentar o software desenvolvido, destacando as funcionalidades implementadas de forma prática e objetiva. Serão apresentados os recursos disponibilizados na aplicação, demonstrando como cada um deles atende aos requisitos levantados e facilita a interação do usuário com o sistema.

9.1 BACK-END

Por meio de sua interface amigável, o Swagger permite que desenvolvedores e usuários técnicos acessem uma documentação detalhada das rotas disponíveis bem como possa executa-las pela ferramenta. Desta forma será apresentado a interface gráfica do mesmo:

Ao acessar o link *"https://lar-certo-api-0f17f4a500cf.herokuapp.com/swagger-ui/index.html#/"*, podemos visualizar a interface gráfica do swagger com 23 endpoints disponibilizados pela API. Na Figura 13, pode ser visto que as funcionalidades são divididas em módulos, desta forma fica fácil e acessivel identificar o propósito de cada funcionalidade do sistema.

Figura 13 – Interface Swagger

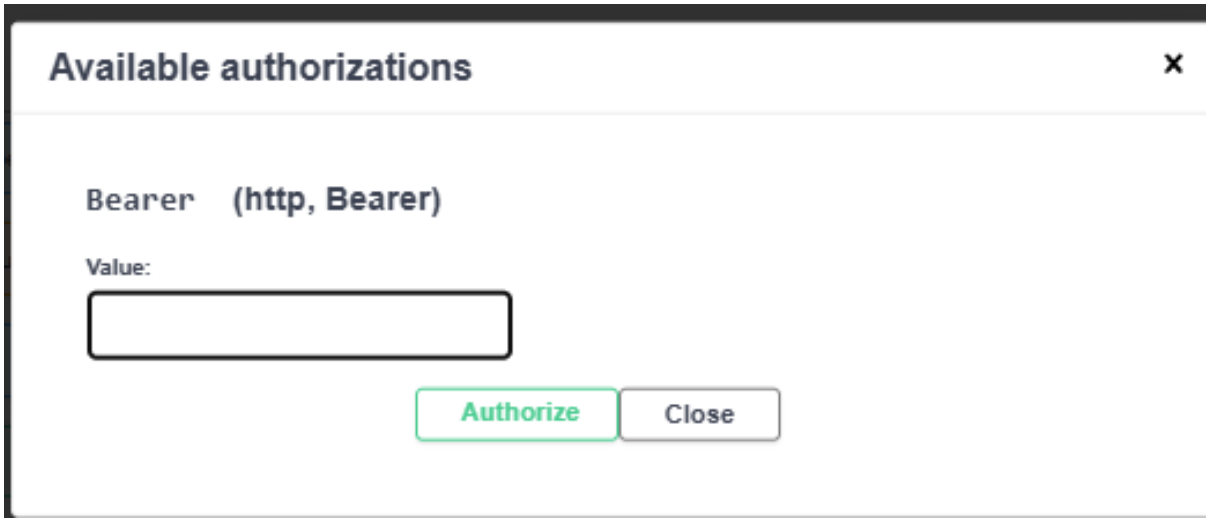


Fonte: autoria própria (2024)

Na Figura 13 pode ser visualizado alguns links logo no início da página no canto superior a esquerda. O primeiro link é o *"/v3/api-docs" que* gera um JSON que pode ser copiado e inserido no site *"https://editor.swagger.io/",* assim um desenvolvedor pode editar o documento gerado de forma online, que pode ser útil para gerar contratos entre times de desenvolvimento que podem usar a documentação para guiar seus desenvolvimentos ou testar funcionalidades da ferramenta. O segundo link "*Contact Anderson Fuhr*", contém o contato do fornecedor da aplicação e por fim o terceiro link "*Repositório Desta API no GitHub*" contém o link com a documentação extra da aplicação, que atualmente direciona ao repositório do GitHub mas pode ser configurável para redirecionar a qualquer página que contenha uma documentação extra do sistema.

Como a aplicação usa autenticação para uso dos recursos é necessário informar ao Swagger um token de autenticação, para isso deve se utilizar os endpoints do módulo de autenticação para fazer login e gera um token JWT, após a geração do token, basta clicar no botão de Authorize no lado direito superior da figura 14. Será então aberto um modal para inserir o token gerado como mostra na figura 14, basta informar o token no campo Value e então o Swagger irá injetar o token em todas as requisições que ele realizar daqui para frente.

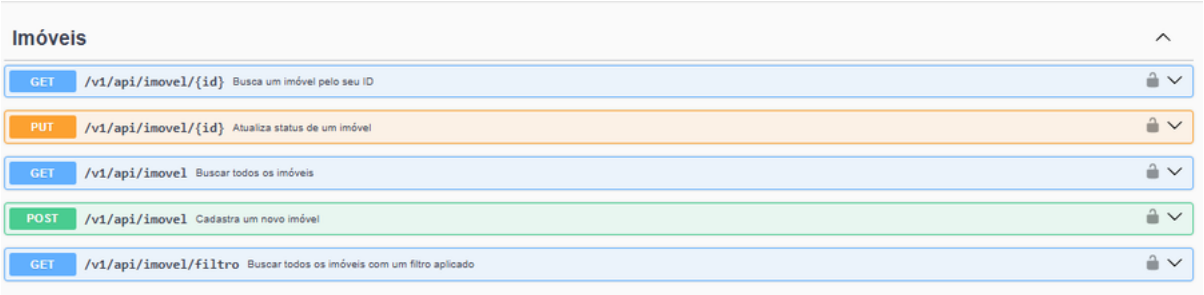
Figura 14 – Swagger - Login



Fonte: autoria própria (2024)

Como visualizado na figura 15, ao clicar para expandir o módulo de imóveis, pode ser visualizado todos os endpoint relacionados a imóveis. Cada endpoint é descrito com seu verbo HTTP, indicando qual ação será realizada, bem como uma breve descrição do endpoint, parâmetros todos configuráveis nas classes de configuração e anotações já mencionados anteriormente.

Figura 15 – Swagger - Módulo Imóveis

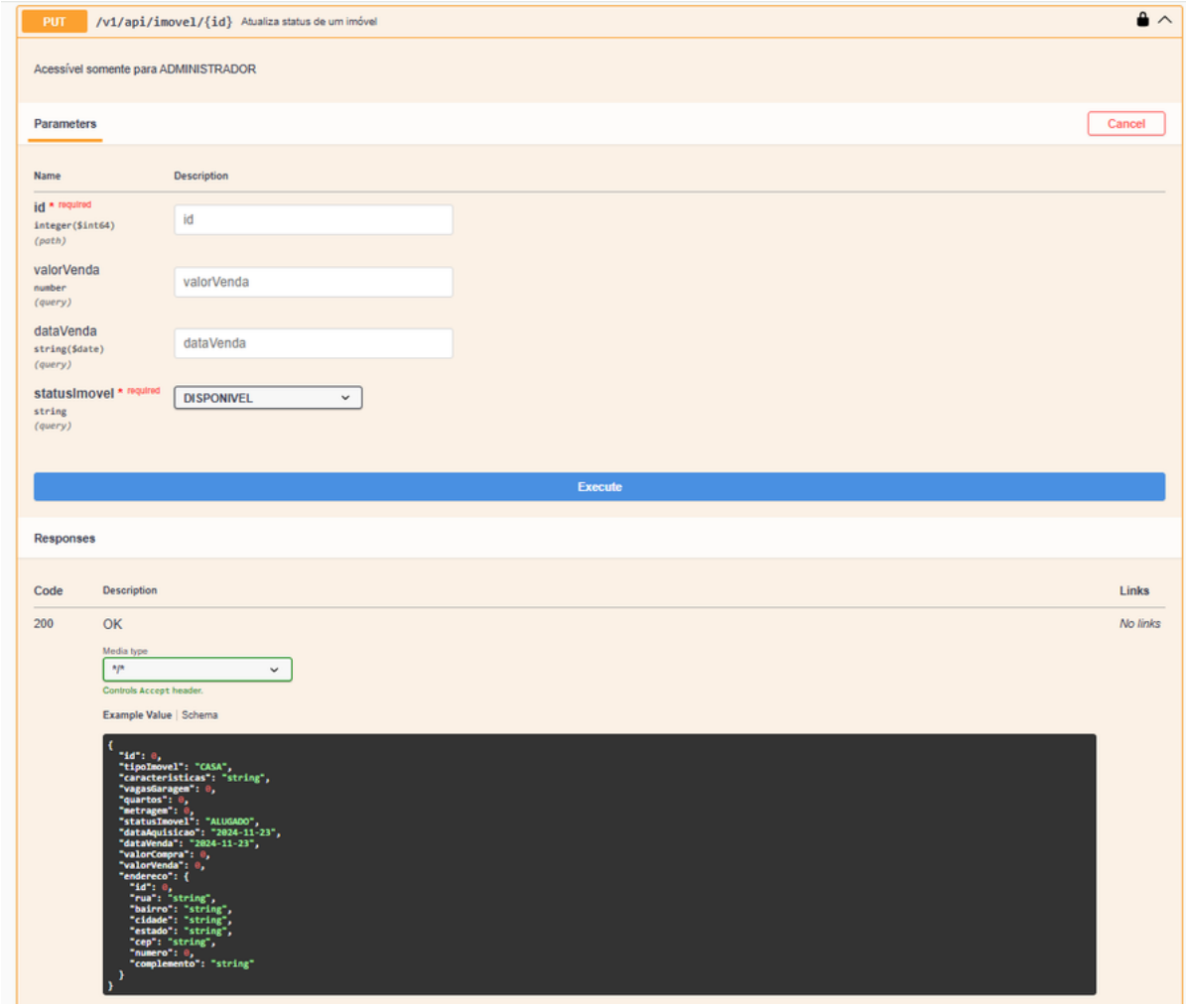


Fonte: autoria própria (2024)

Na Figura 16, é possível observar que, ao clicar em um dos recursos que o usuário deseja consumir, o contrato a ser seguido para o envio e o retorno da requisição HTTP é exibido de forma detalhada. No exemplo mostrado, a requisição utiliza o método HTTP PUT na URL *"/v1/api/imovel/{id}" com a seguinte descrição dos campos a serem enviados:*

- O *id* é um parâmetro obrigatório do tipo inteiro, que deve ser fornecido como parte da URL.
- O campo *valorVenda* é um parâmetro opcional do tipo decimal, enviado como query param na URL.
- A *dataVenda* também é documentada como opcional e deve ser enviada no formato de data como query param .
- Já o campo *statusImovel* é um parâmetro obrigatório, enviado como query param na URL, e deve assumir apenas os valores permitidos, previamente definidos e listados na documentação.

Figura 16 – Swagger - Exemplo documentação de um endpoint

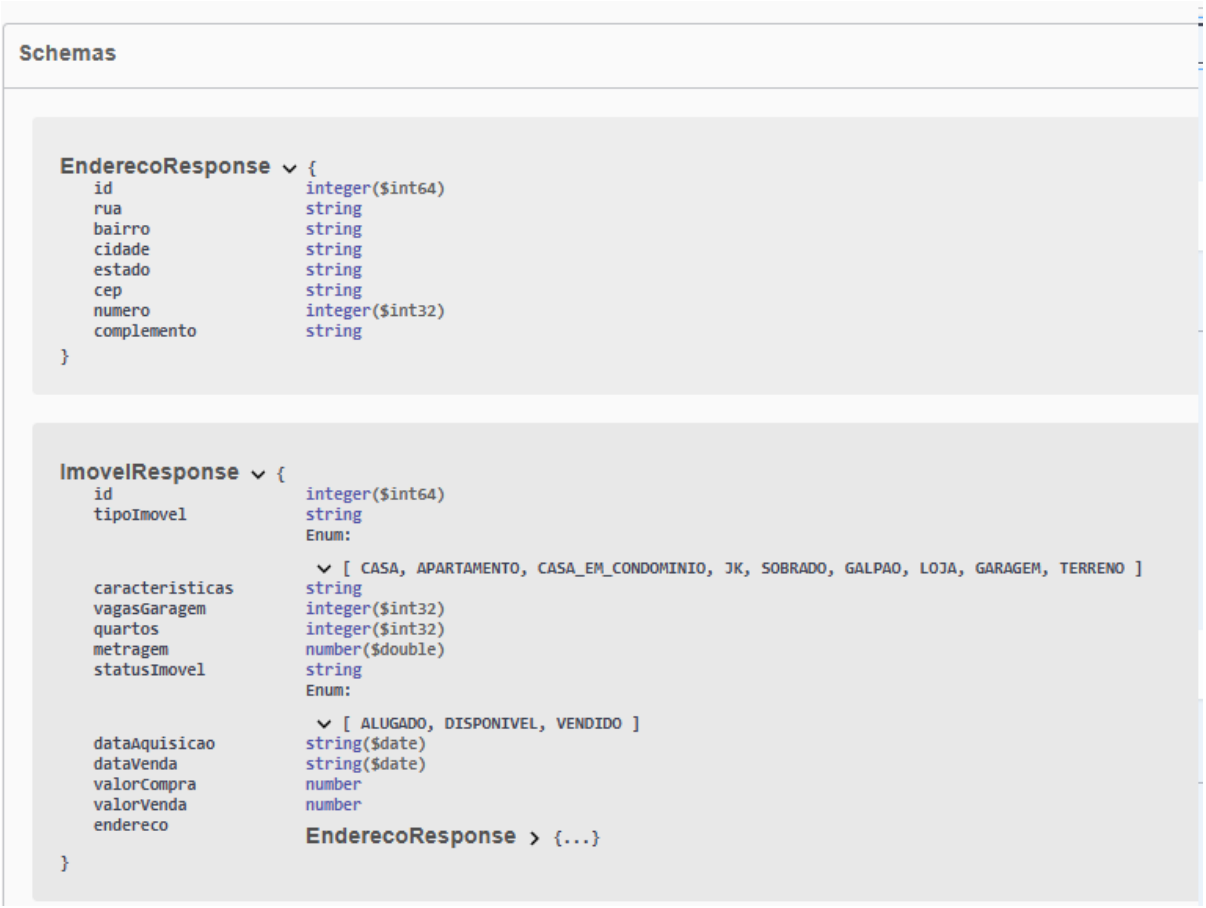


Fonte: autoria própria (2024)

Como visualizado na Fiigura 16 é possível clicar no botão azul “Execute” para executar a requisição HTTP inserindo os dados necessários, e conforme visualizado no canto esquerdo inferior há o contrato do JSON que será retornado na requisição bem como os possíveis status HTTP que podem ser retornados.

O módulo Schemas no Swagger é uma seção dedicada a descrever os modelos de dados utilizados pela API nas requisições e respostas. Ele serve como uma referência para os desenvolvedores, apresentando a estrutura e os detalhes de cada objeto, além de exemplos claros para facilitar a compreensão e implementação como demonstrado na figura 17.

Figura 17 – Swagger - Módulo Schema



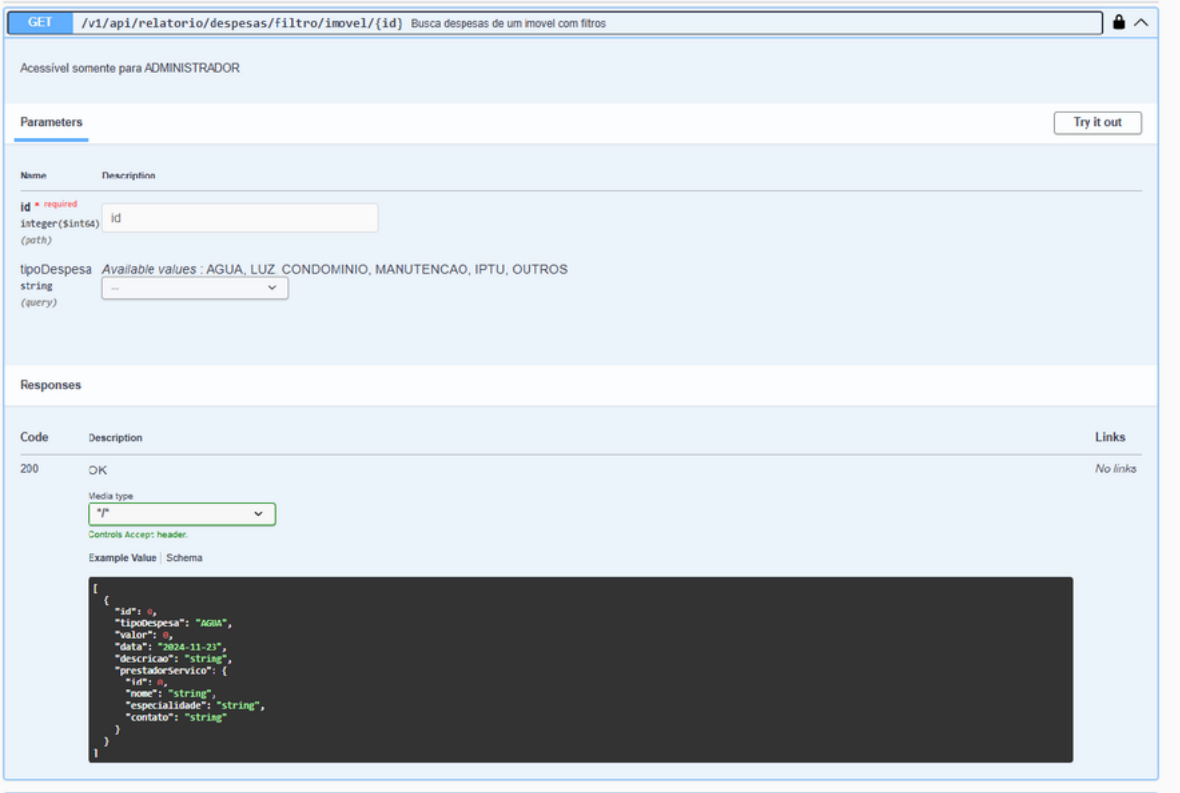
Fonte: autoria própria (2024)

Embora a aplicação já tenha diversos endpoints documentados e implementados no back-end, alguns deles ainda não foram integrados ao front-end. Esses endpoints, apesar de funcionais e devidamente testados, não estão sendo atualmente consumidos pela interface do usuário. Isso se deve ao fato de que, no momento, o foco esteve na entrega das funcionalidades essenciais do MVP, priorizando os casos de uso mais críticos para a gestão imobiliária. Será descrito a seguir os endpoint que não foram utilizados no front-end, mas que podem trazer significativo valor ao uso pelo usuário.

No total foram 6 endpoints que não foram implementados no front-end, aos quais são descritos a seguir:

Endpoint de relatório "*GET /v1/api/relatorio/despesas/filtro/imovel/{id}*": busca todas as despesas de um imóvel específico, podendo filtrar por uma despesa como água, luz ou qualquer outra despesa no geral. Como mostra na figura 18, é possível enviar o Id do imóvel que deseja consultar e passar o tipo de despesa ao qual quer filtrar, se deixar o tipo de despesa em branco irá retornar todas as despesas do imóvel. Este recurso pode ser muito útil para entender qual o maior gasto de imóvel específico.

Figura 18 – Swagger - Endpoint relatório despesa



Fonte: autoria própria (2024)

Endpoint de relatório “GET /v1/api/relatorio/despesa/imovel/{id}”: busca uma lista de soma de despesas por ano e mês de um imóvel passando o Id do imóvel como parâmetro. O retorno da chamada irá resultar em uma lista no formato JSON, onde será apresentado o total de gastos com despesas de cada mês por ano como mostra no JSON abaixo. Estes dados podem ser muito úteis para visualizar qual os meses que houveram maior gasto com o imóvel.

```
[
  {
    "total": 500,
    "ano": 2024,
    "mes": 1
  },
  {
    "total": 400,
    "ano": 2024,
    "mes": 2
  },
  {
    "total": 350,
    "ano": 2024,
    "mes": 3
  }
]
```

Fonte: autoria própria (2024)

Endpoint “GET /v1/api/relatorio/aluguel/{id}”: este recurso constrói um relatório com a soma total ganha e prevista com aluguém de um determinado imóvel. O endpoint recebe um

Id de um imóvel como parâmetro e retorna uma lista com os valores somados em formato JSON como demonstrado no JSON a seguir.

```
[
  {
    "ano": 2024,
    "valorArrecadado": 1200
  },
  {
    "ano": 2025,
    "valorArrecadado": 7200
  },
  {
    "ano": 2026,
    "valorArrecadado": 7200
  },
  {
    "ano": 2027,
    "valorArrecadado": 6600
  }
]
```

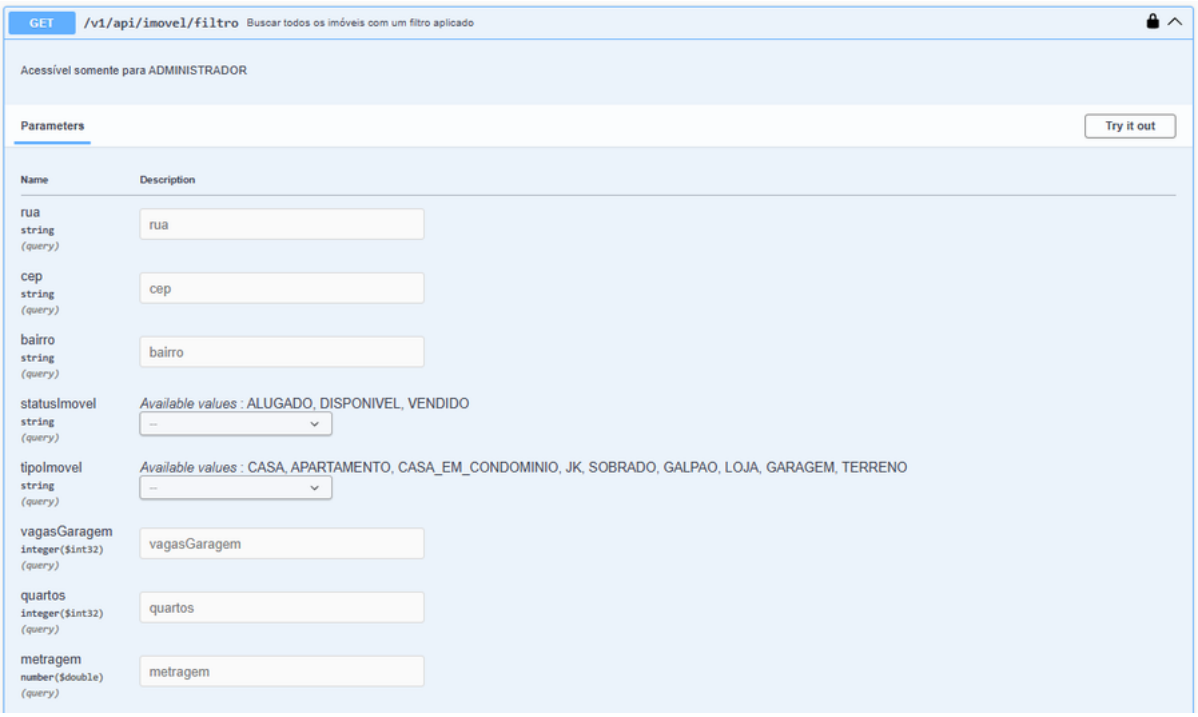
Como pode ser visualizado acima, o JSON retornado pode ser utilizado tanto para vizualização de ganhos totais de um imóvel em um determinado ano como fazer possiveis projeções de arrecadação.

Endpoint despesa “POST /v1/api/despesa”: este recurso tem como finalidade cadastrar uma despesa à um imóvel, que pode ser de vários tipos como AGUA, LUZ, CONDOMINIO, MANUTENCAO, IPTU ou OUTROS.

Endpoint de prestador de serviço “GET /v1/api/prestador-servico/{nome}”: tem como finalidade buscar um possível prestador de serviço pelo nome, e será utilizado futuramente em uma tela para cadastro de despesa do tipo manutenção, onde será exigido que haja um prestador de serviço responsável pela manutenção.

Endpoint de imóvel “GET /v1/api/imovel/filtro”: este endpoint buscar uma lista paginada de imóveis baseado em diversos filtros como demonstrado na Figura 19. Este método é muito útil para que o usuário possa no futuro escolher exatamente quais imóveis quer visualizar, filtrando desde endereço desejado até número de cômodos ou tipo de imóvel, porém, por sua alta complexidade na implementação, tanto do back-end quanto no front-end, não houve tempo hábil para seu uso na front-end.

Figura 19 – Swagger - Filtro imóveis



Fonte: autoria própria (2024)

9.2 FRON-TEND

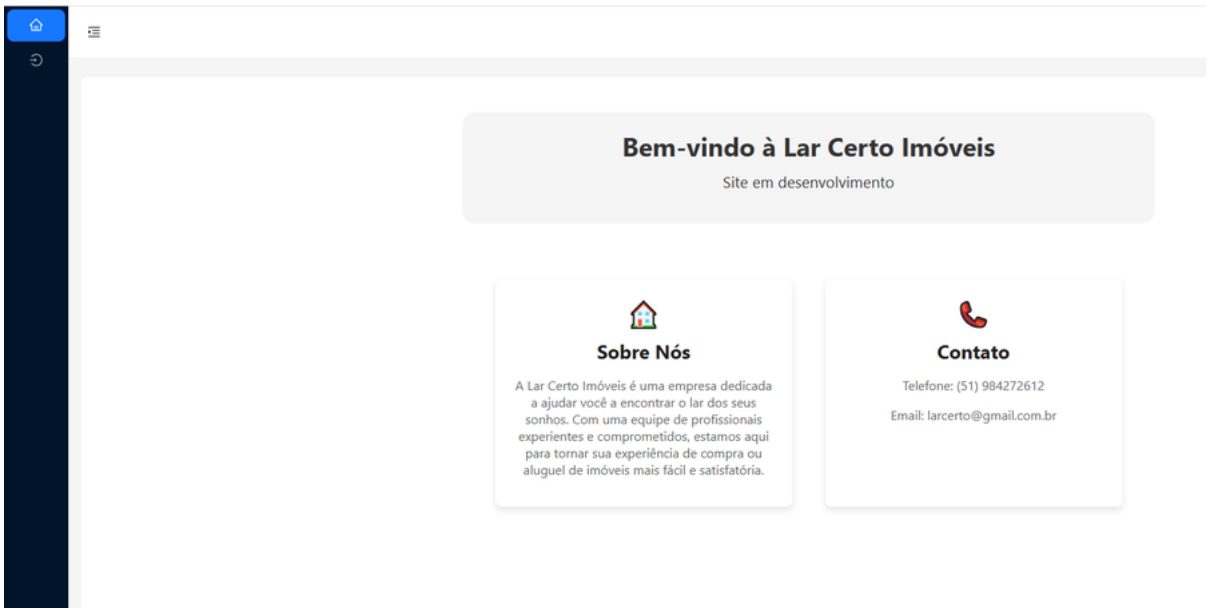
O layout da aplicação front-end, desenvolvido em React, foi construído utilizando a biblioteca de componentes Ant Design(ANT DESIGN, 2024). Essa biblioteca foi escolhida por sua ampla gama de componentes pré-desenvolvidos, estilizados de forma moderna e responsiva, agilizando o desenvolvimento e garantindo uma experiência de usuário consistente e intuitiva.

A página inicial do site Lar Certo, apresentada na Figura 20, possui um layout simples e funcional, projetado para facilitar o acesso às principais funcionalidades da plataforma. O site conta com uma interface responsiva e intuitiva, proporcionando uma experiência agradável para os usuários.

No lado esquerdo da página, há um menu retrátil que, após o login do usuário, exibe as opções de acesso disponíveis conforme o perfil do usuário.

Atualmente, o sistema possui apenas o perfil de ADMINISTRADOR, mas está projetado para ser expandido futuramente, permitindo a criação de novos perfis e controles de acesso personalizados para diferentes funcionalidades, tanto para as já existentes quanto para as que possam ser adicionadas. Essa estrutura garante flexibilidade e escalabilidade à medida que a plataforma evoluir.

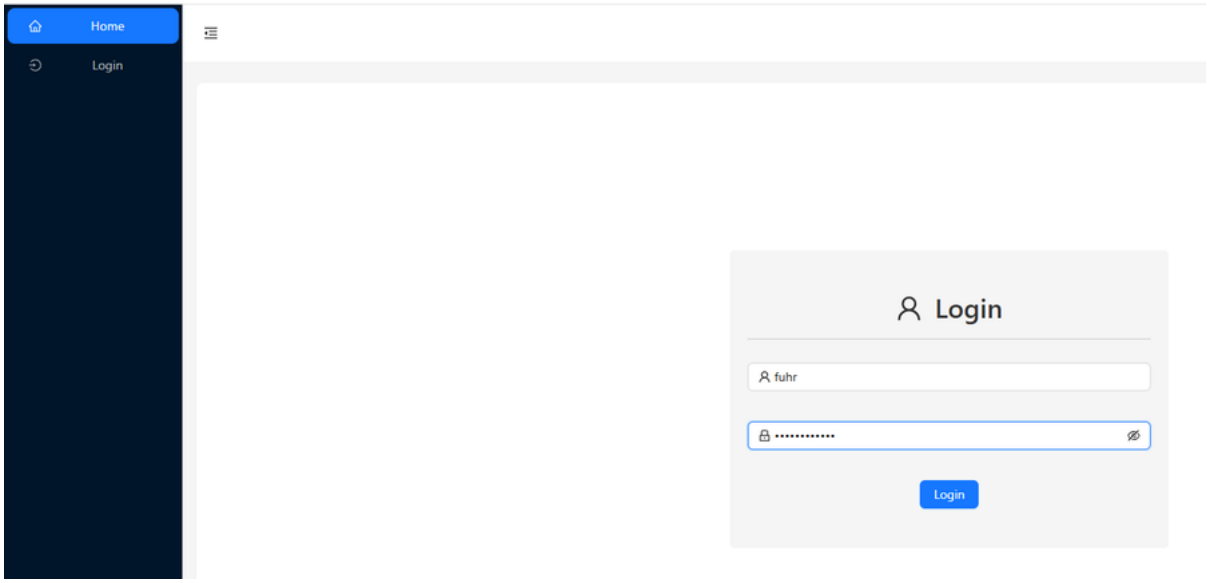
Figura 20 – Página inicial



Fonte: autoria própria (2024)

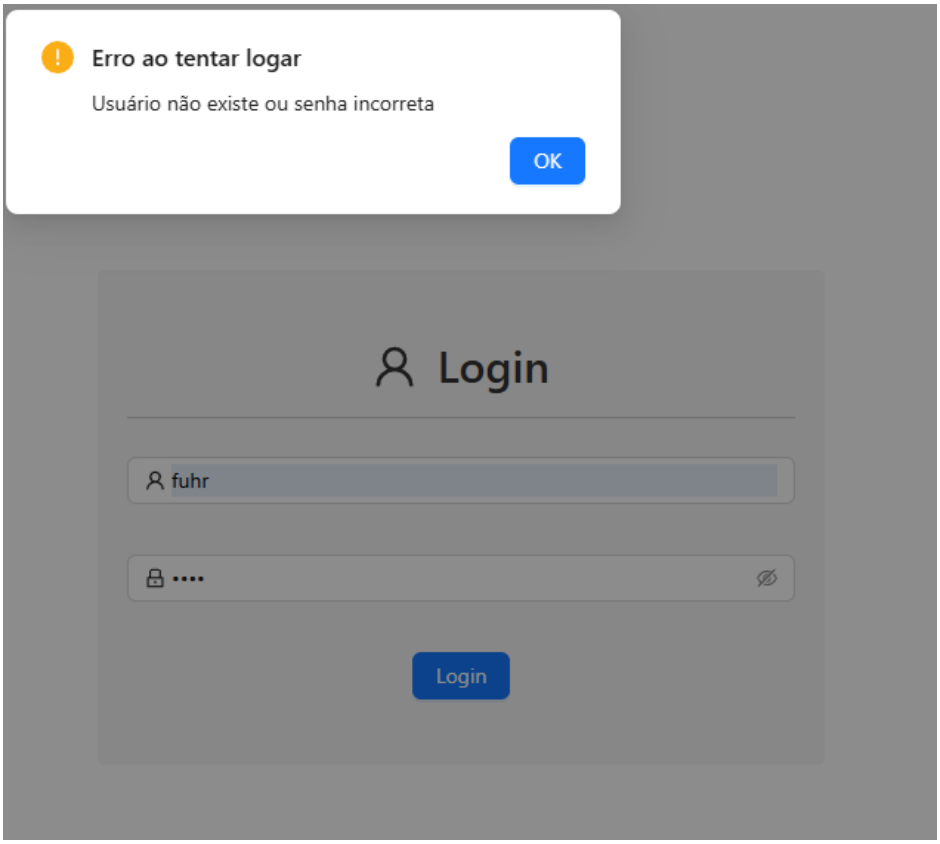
Caso o usuário digite uma rota privada diretamente na URL do navegador, o mesmo será redirecionado ao login, como mostra a figura 21. Nesta tela o usuário deverá informar o nome de usuário e senha cadastrados no sistema. No campo onde a senha poderá ser informada o usuário pode optar por deixar a senha visível ou não, clicando no ícone de olho ao lado direito do campo. Após clicar no botão azul escrito L*ogar*, um loading será lançado na tela até que a confirmação do back-end seja retornada, caso haja erro, um pop-up com um alerta será lançado, conforme demonstrado na figura 22. Caso o usuário não preencha todos os campos, um alerta será lançado logo abaixo dos campos como mostra na figura 23, este comportamento irá se seguir em todas as telas onde haja preenchimento obrigatório.

Figura 21 – Login



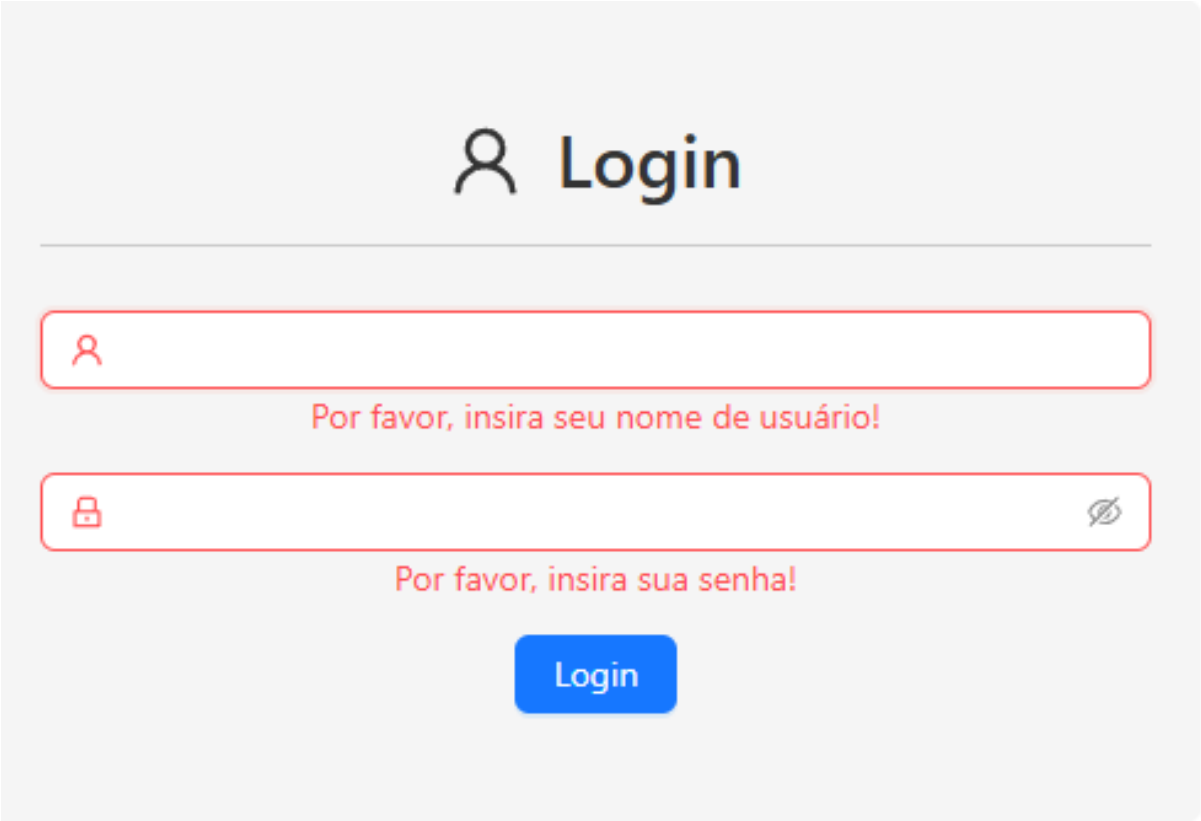
Fonte: autoria própria (2024)

Figura 22 – Login - Poo-up alerta



Fonte: autoria própria (2024)

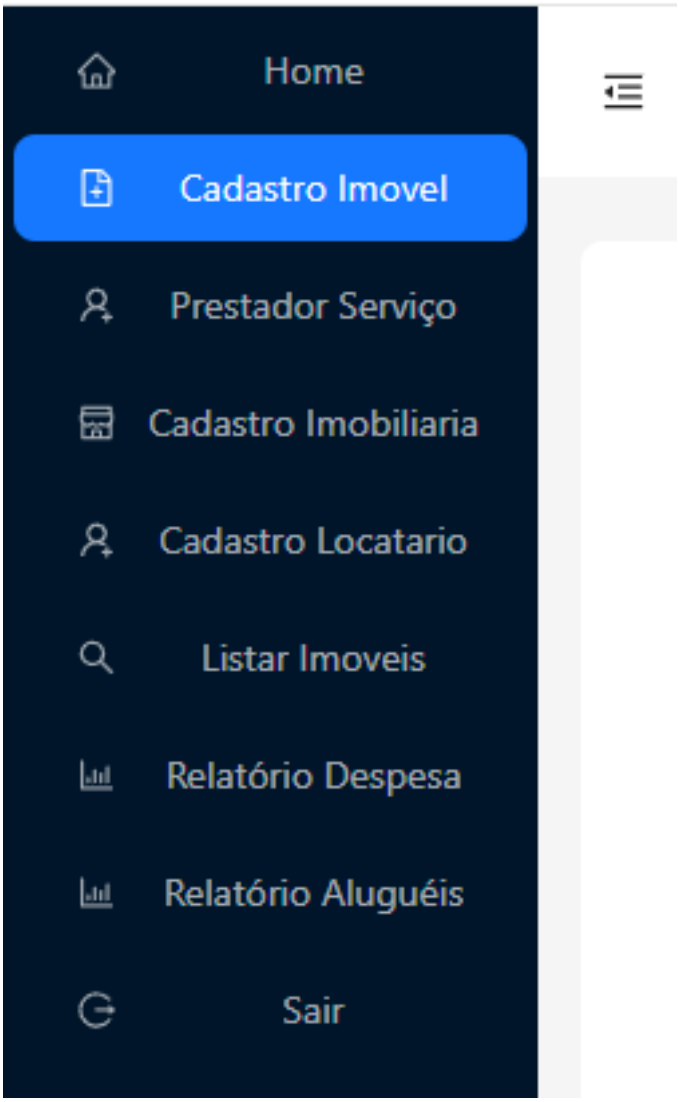
Figura 23 – Login - Alerta erro



Fonte: autoria própria (2024)

Se o login for bem-sucedido o suário em fim terá acesso ao menu de ações conforme mostra a Figura 24. Este menu pode ser personalizável no futuro, disponibilizando menus conforme o perfil do usuário logado.

Figura 24 – Menu - Visão administrador logado

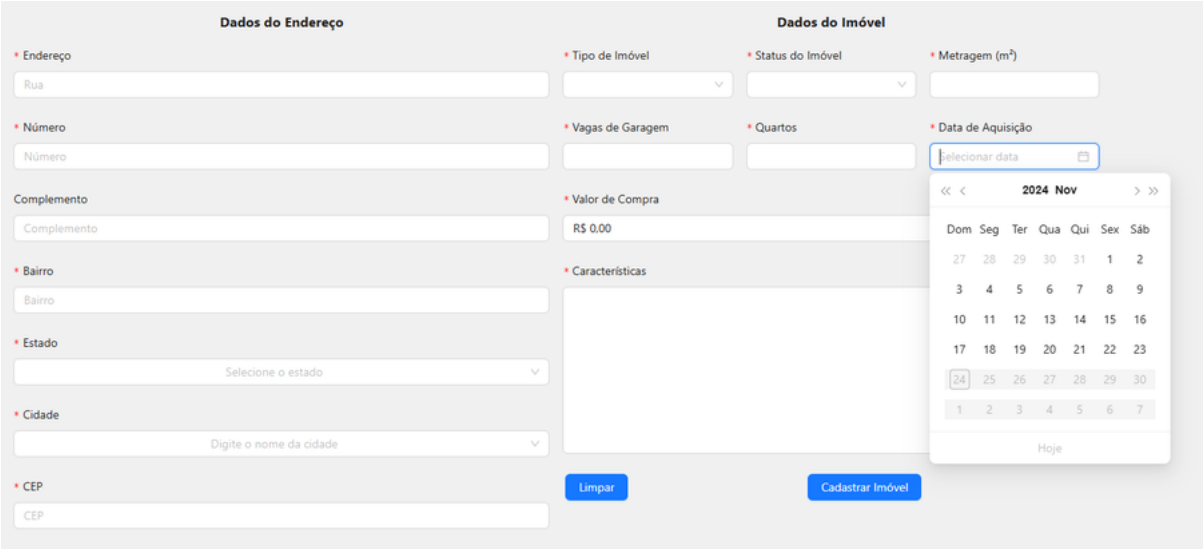


Fonte: autoria própria (2024)

A Figura 24 apresenta a tela de cadastro de imóvel, onde todos os campos marcados com um asterisco vermelho são obrigatórios. Caso algum desses campos não seja preenchido, o sistema impedirá a realização do cadastro e exibirá um alerta em vermelho, semelhante ao mostrado na tela de login na Figura 26. Esse comportamento tem como objetivo orientar o usuário sobre as informações necessárias para concluir o cadastro corretamente.

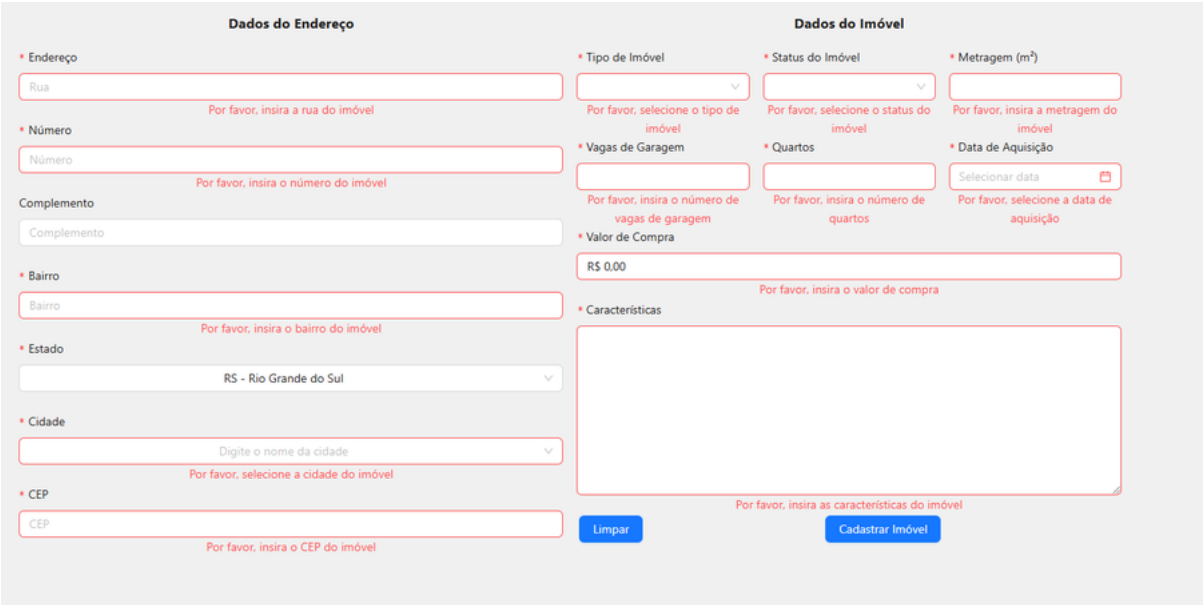
Na parte inferior da tela, ao centro, há um botão azul com o texto ***"Limpar"***, cuja função é resetar o formulário de cadastro, permitindo que o usuário recomece o preenchimento de forma rápida e prática. Essa funcionalidade contribui para melhorar a experiência do usuário ao interagir com o sistema.

Figura 25 – Cadastro Imóvel



Fonte: autoria própria (2024)

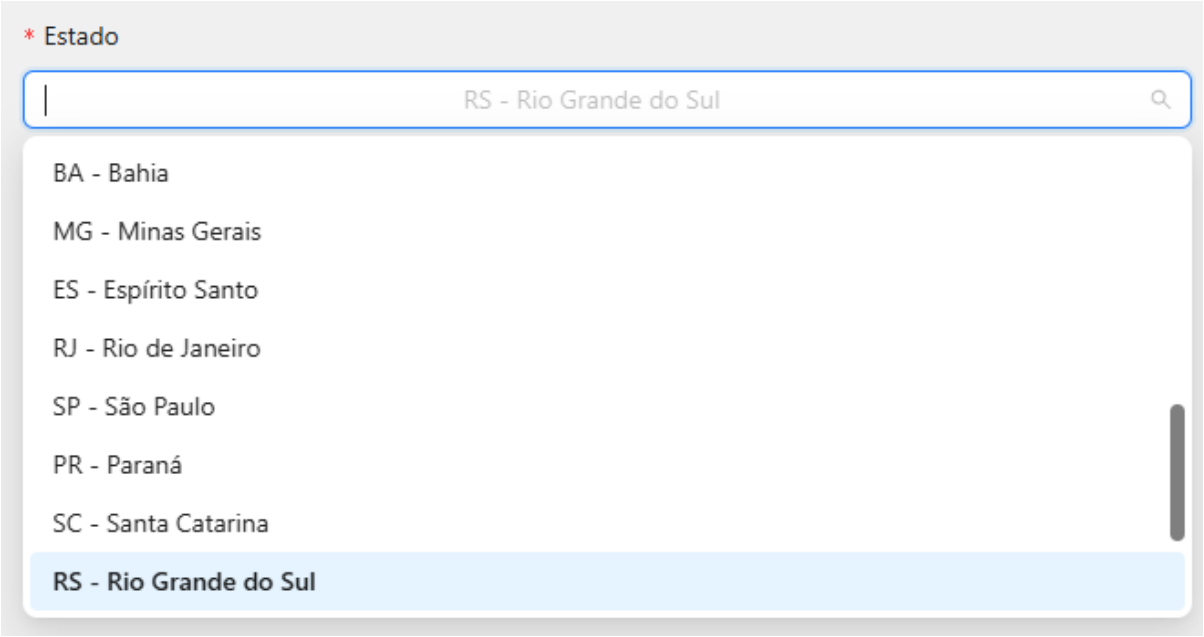
Figura 26 – Cadastro Imóvel - Alerta erro



Fonte: autoria própria (2024)

Na mesma tela de cadastro de imóvel demonstrada na Figura 25, o campo de Estado e Cidade, apresentam uma lista demonstrada na Figura 27, onde é feito o uso da API do IBGE para popular às duas listas de estados e cidades. Para selecionar o Estado o front-end usa a API do IBGE e popula alista, após seleção do estado pelo usuário, o sistema usa o nome do estado selecionado para novamente buscar a lista das cidades deste estado, trazendo comodidade ao usuário que terá acesso à lista de todos os estados do Brasil.

Figura 27 – Cadastro imóvel - Input do Estado



Fonte: autoria própria (2024)

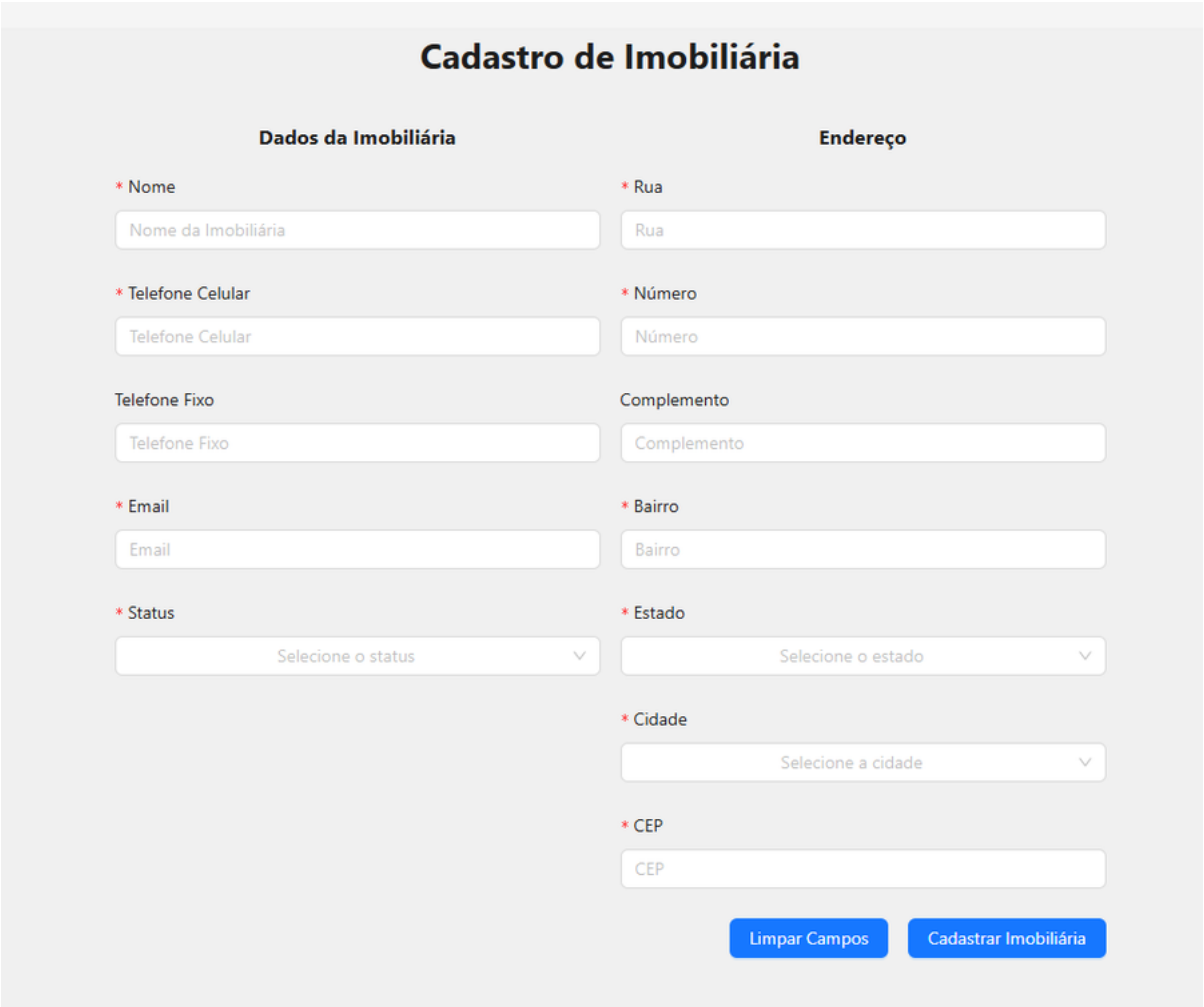
Para cadastrar um prestador de serviço basta acessar a opção “Prestador Serviço” como consta no menu da Figura 24 e o usuário terá acesso ao formulário de cadastro de um prestador de serviço conforme visualizado na Figura 28. O cadastro segue com o mesmo modelo de usabilidade do cadastro de imóvel demonstrado na figura 25.

Figura 28 – Formulário de cadastro prestador de serviço

Fonte: autoria própria (2024)

Ao acessar o menu “Cadastro Imobiliária” conforme consta na Figura 24, o sistema irá redirecionar para a tela de cadastro de imobiliária demonstrada na figura 29 onde é possível cadastrar uma imobiliária, que será posteriormente utilizada para realizar o cadastro de um aluguel caso o mesmo tenha sido realizado com intermédio de um agende imobiliário. Os seguem o layout e usabilidade conforme as demais telas, inclusive com o uso da API de IBGE para os campos de Estado e Cidade.

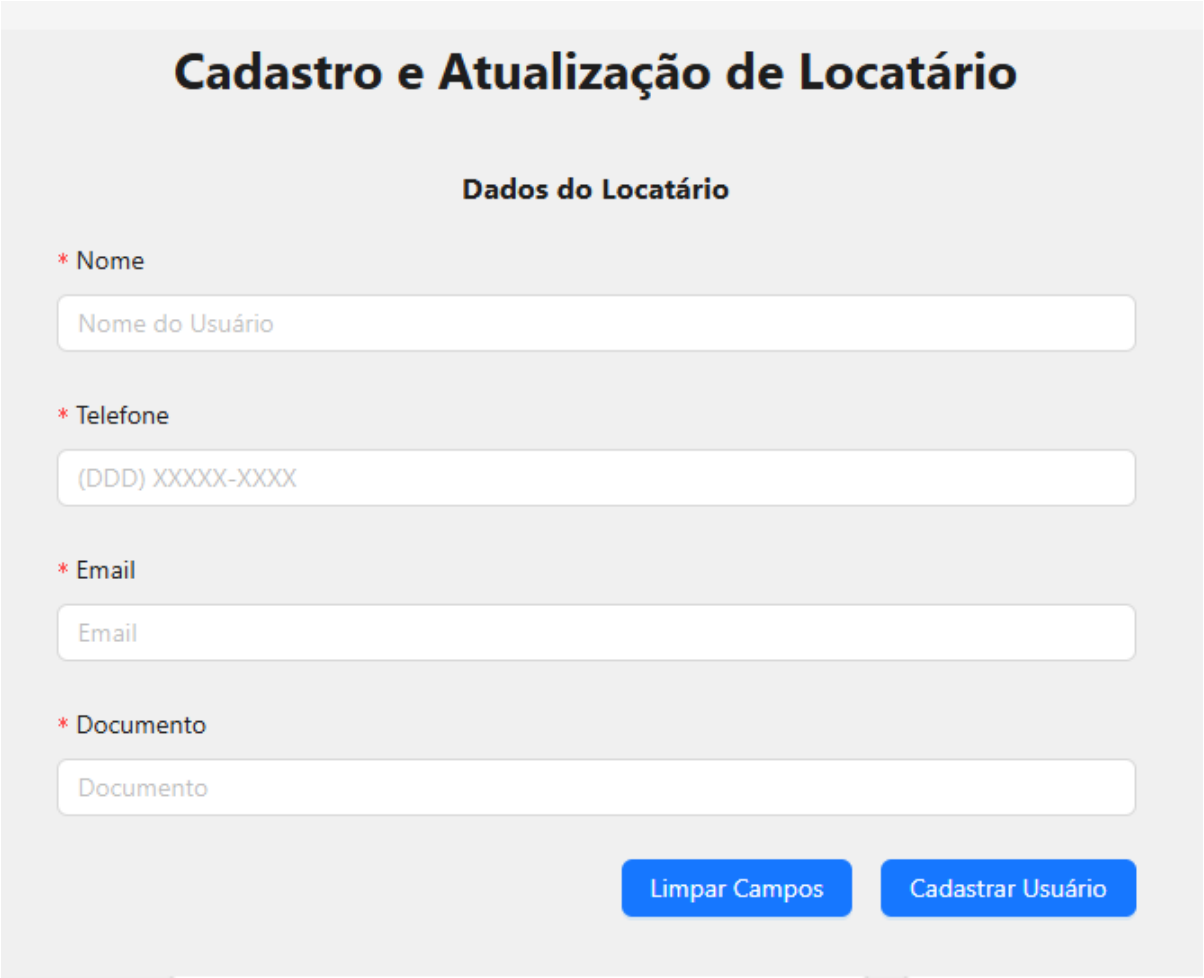
Figura 29 – Formulário cadastro imobiliária



Fonte: autoria própria (2024)

Acessando o menu de “Cadastro Locatário” conforme consta na Figura 24, o sistema irá redirecionar para o formulário conforme Figura 30. Nesta tela caso o CPF digitado já conste no cadastro, o mesmo terá seus campos atualizados, sendo assim, esta tela permite tanto o cadastro, quanto o update do locatário já inserido no sistema.

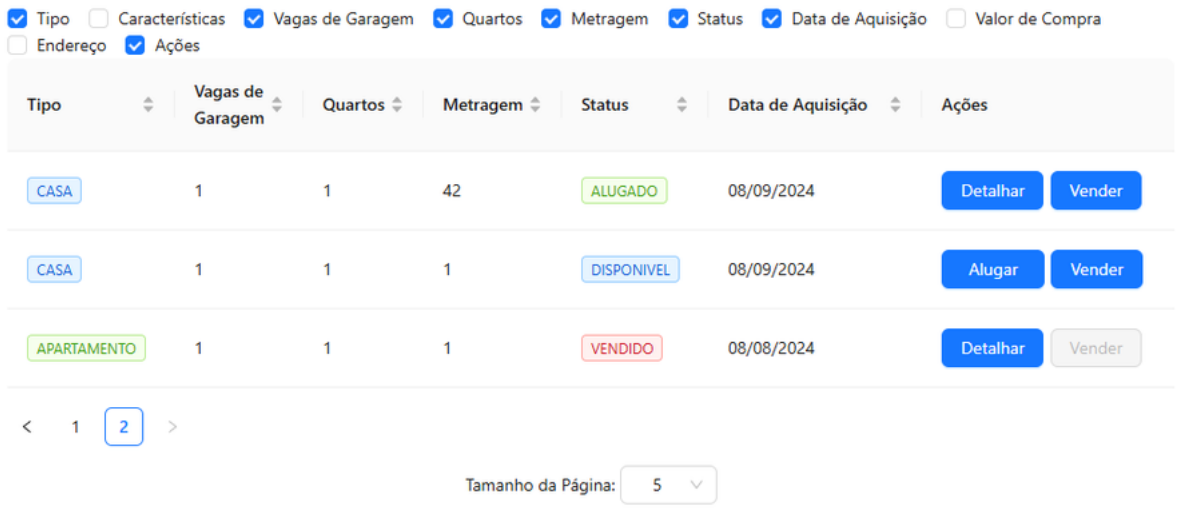
Figura 30 – Formulário de cadastro de locatário



Fonte: autoria própria (2024)

Ao acessar menu “Listar Imóveis” da figura 24 o usuário terá acesso à lista paginada e detalhada de imóveis, como mostra a Figura 31. Na parte superior da tela, contém caixas de seleção onde o usuário pode personalizar as colunas que devem ou não serem mostradas na tabela. A tela também separa por cores os tipos de imóveis e o status do mesmo. Caso o usuário deseje ordenar a tabela por uma das colunas, basta clicar no cabeçalho da coluna e o mesmo poderá ordenar pela coluna selecionada. A tela faz uma busca paginada, onde o usuário pode escolher a quantidade de itens demonstrados na tabela, fazendo a seleção através do input *“Tamanho da Página”* localizado no centro inferior da página. No canto esquerdo inferior é possível navegar entre as páginas.

Figura 31 – Listagem e manipulação imóveis

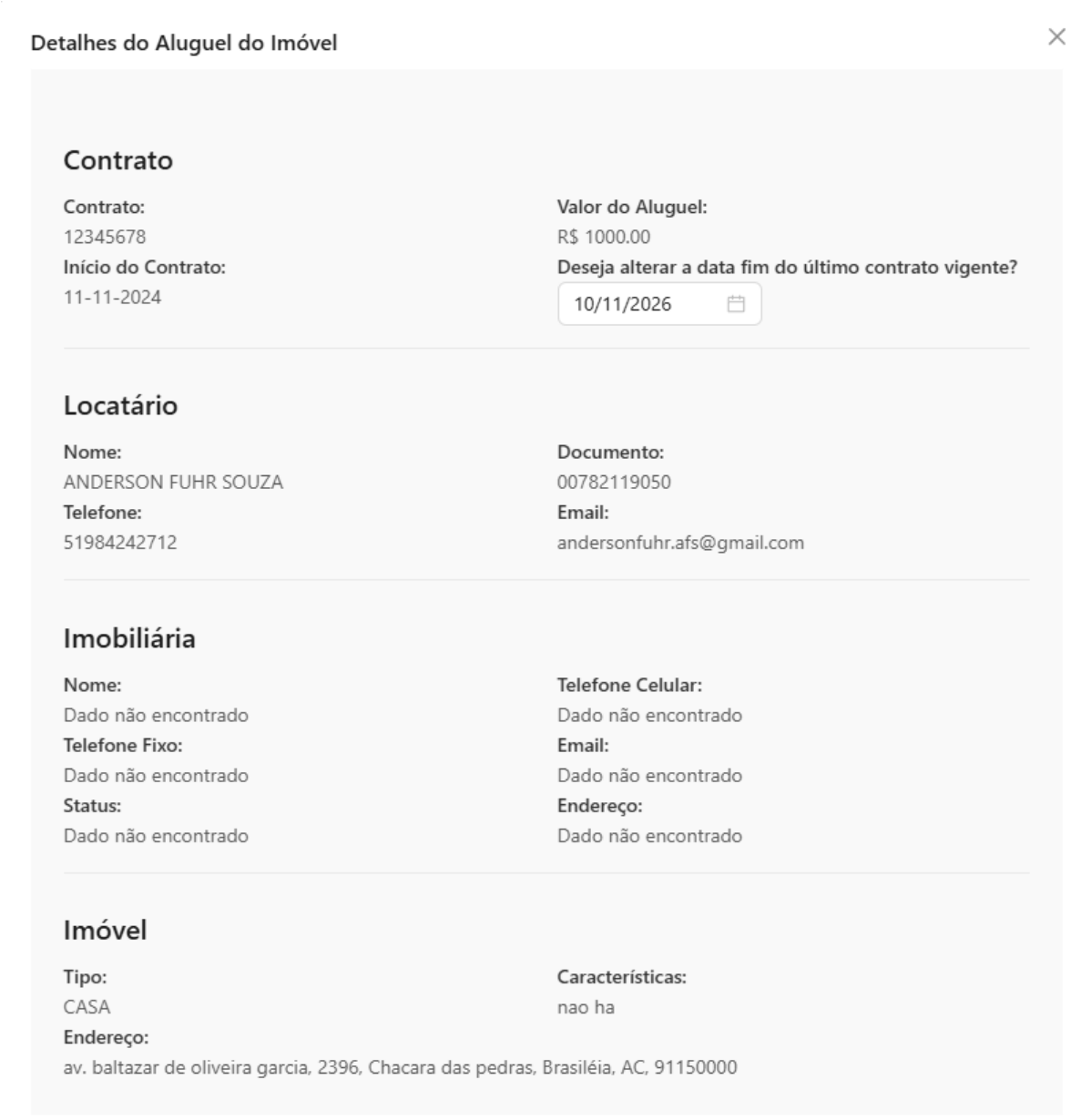


Fonte: autoria própria (2024)

Na Figura 31, dentro da linha de informação do imóvel na tabela pode ser visualizado botões de ações no lado direito, visualizados conforme o status de um imóvel. Caso o imóvel esteja com status de ALUGADO, somente os botões de “Detalhar” e “Vender” serão visualizados. Caso o imóvel contenha o status de DISPONIVEL, somente os botões de “Alugar” e “Vender” serão disponíveis ao usuário. Caso o status do imóvel seja VENDIDO, somente o botão de “Detalhar” estará acessível ao usuário.

Ao clicar no botão “Detalhar” da tela demonstrada na Figura 31, será aberto o modal de informações sobre o imóvel, onde também é possível editar a data final da vigência de um aluguel como mostra na figura 32. Nesta tela da Figura 32, há um pedido que o campo *“Valor de Aluguel”* também seja um campo editável, melhoria esta que vai ajudar na atualização dos valores que são anualmente reajustados por índices de mercado.

Figura 32 – Modal detalhes de um imóvel



Fonte: autoria própria (2024)

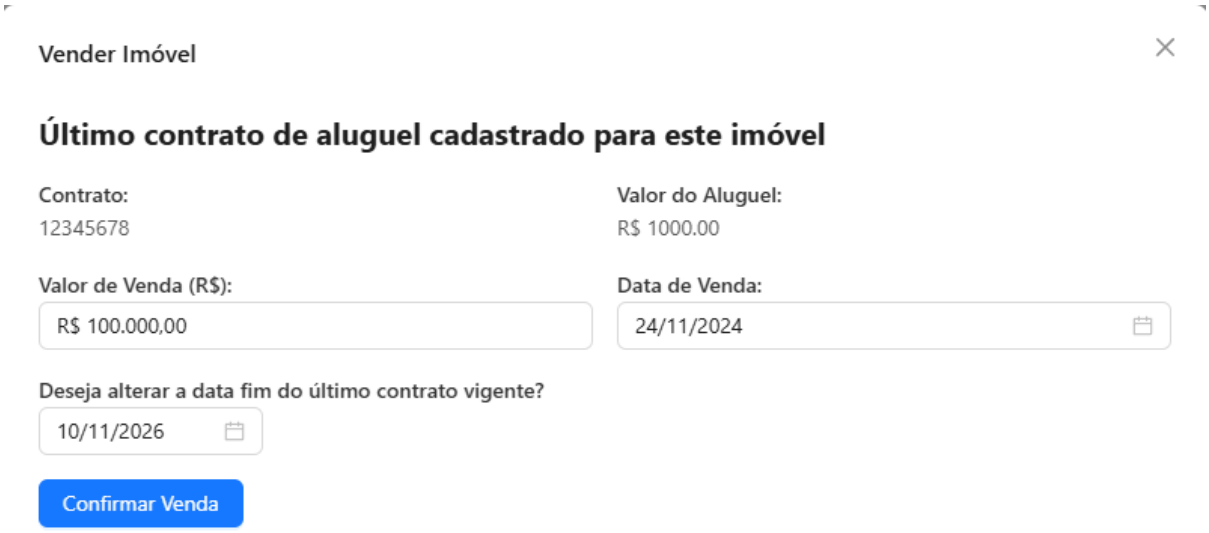
Ao clicar no botão de *"Alugar"* um imóvel como mostra a tela na Figura 31, o sistema abre um modal mostrado na figura 33 onde o usuário pode preencher um formulário de cadastro de um novo aluguel para o imóvel. O único campo não obrigatório é de *"Imobiliaria"*, que ao selecionar o usuário tem acesso a uma lista de imobiliárias já cadastradas no sistema. Como atualmente existem poucas imobiliárias prestando serviço, não se foi necessário um campo de busca, porém é uma futura melhoria que pode ser realizada neste campo, tornando-o mais extensível para caso no futuro haja muitas imobiliárias cadastradas, melhorando assim a experiência do usuário.

Figura 33 – Modal cadastro aluguel

Fonte: autoria própria (2024)

Caso o usuário deseje vender um imóvel, ele não será retirado da lista ou deletado, para salvar seu histórico. Ao clicar no botão de “Vender” na tela demonstrada na Figura 31, um modal demonstrado na Figura 34 irá mostrar os dados do último aluguel cadastrado para o imóvel e permitir que seja feita a edição do status do imóvel para vendido. Caso o imóvel contenha um aluguel vigente, o usuário terá a possibilidade de alterar a data fim do contrato, clicando no input de data ao lado esquerdo inferior do modal, caso não haja contrato vigente, será aberto um modal apenas para informar o valor de venda e data da venda como mostra a figura 35. Após preencher os campos o usuário será alertado que esta ação não permite ser desfeita como mostra na Figura 36.

Figura 34 – Modal Venda com contrato aluguel vigente



Fonte: autoria própria (2024)

Figura 35 – Modal Venda sem contrato aluguel

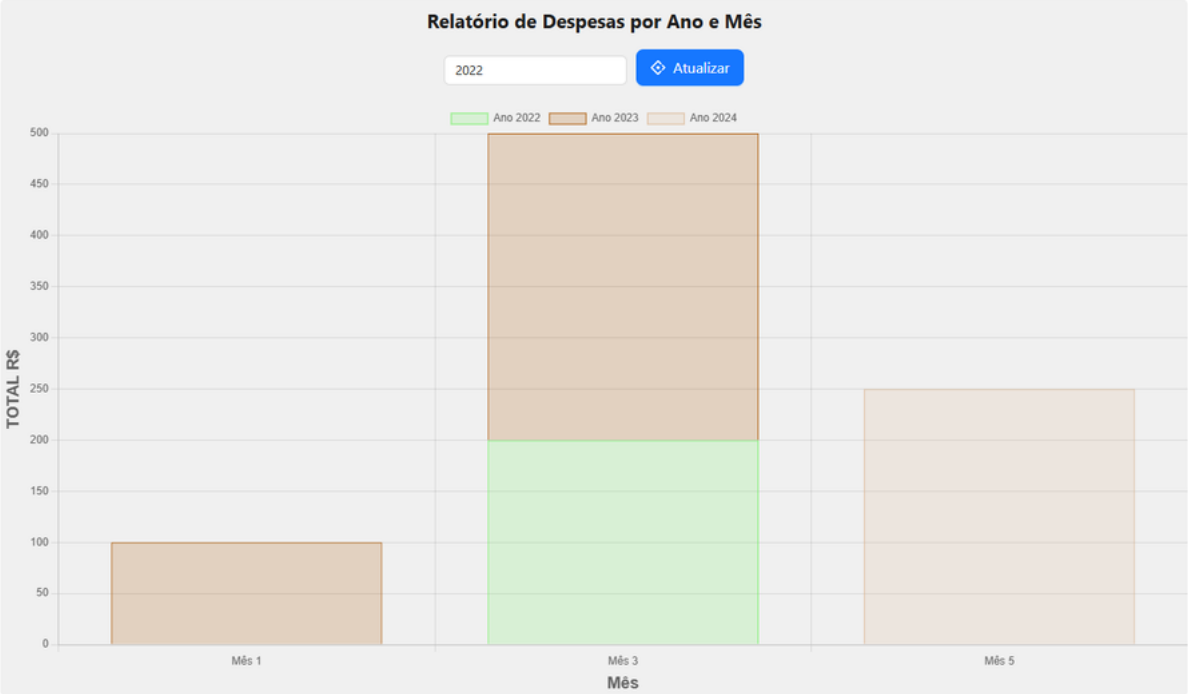
Fonte: autoria própria (2024)

Figura 36 – Modal Venda - alerta ação

Fonte: autoria própria (2024)

O software apresenta relatórios em formato de gráficos. Ao acessar o menu principal, ilustrado na Figura 24, e selecionar a opção “Relatório Despesa”, o usuário será direcionado ao gráfico mostrado na Figura 37. Nesse gráfico, é possível selecionar o ano inicial desejado para visualizar os dados, inserindo o ano no campo de entrada localizado no centro superior da tela. Após clicar no botão “Atualizar”, os dados serão exibidos. Cada ano é representado por uma cor específica, com os meses dispostos no eixo X e a soma total das despesas relacionadas a todos os imóveis no eixo Y. Ao posicionar o mouse sobre uma das cores do gráfico, o valor total do mês correspondente naquele ano será exibido. Caso deseje, o usuário pode ocultar a exibição de um ou mais anos no gráfico clicando nos itens da lista de anos, localizada acima do gráfico, no centro da tela.

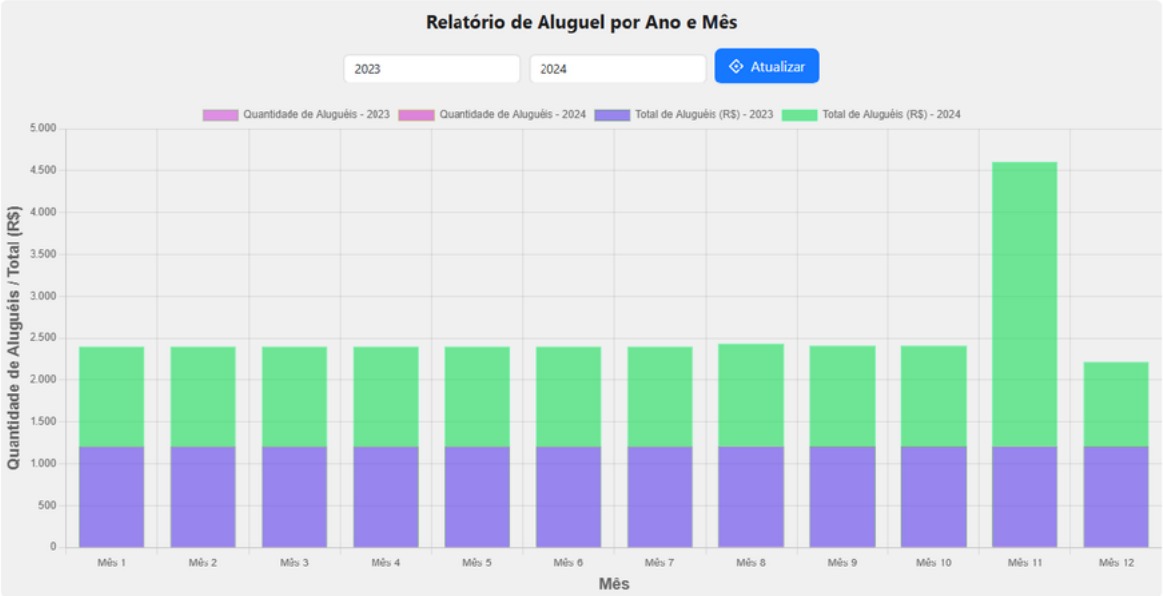
Figura 37 – Relatório despesa



Fonte: autoria própria (2024)

A Figura 38 apresenta um relatório de rendimentos de aluguéis, acessível através do menu principal, mostrado na Figura 24, ao clicar na opção “Relatório Aluguéis”. Nesse relatório, o eixo X exibe os valores mensais arrecadados com aluguéis, enquanto no eixo Y o usuário pode optar por visualizar a quantidade de contratos de aluguel realizados por mês ou a soma total dos contratos. Assim como no relatório de despesas (Figura 37), o usuário pode selecionar quais anos ou dados serão exibidos no gráfico, ajustando a visualização de acordo com sua necessidade.

Figura 38 – Relatórios aluguéis



Fonte: autoria própria (2024)

10 CONCLUSÃO

A implementação de práticas robustas de qualidade e escalabilidade no desenvolvimento deste sistema foi essencial para garantir um código confiável e preparado para o crescimento futuro. A construção de uma estrutura completa de testes, contemplando desde testes unitários até E2E na API, assegurou a verificação contínua das regras de negócio. Verificou-se que o uso de uma pipeline de CI/CD, configurada para garantir 80% de cobertura mínima de testes automatizados, junto ao JaCoCo, adicionou um nível extra de segurança, permitindo que as mudanças no código fossem avaliadas e testadas antes de serem integradas ao ambiente de produção.

Além disso, as ferramentas como ESLint e Sonar desempenharam um papel crucial no monitoramento e na análise da qualidade do código. O ESLint foi essencial para desenvolver o código do front-end conforme boas práticas e padrões de formatação, enquanto o SonarLint realizou uma análise profunda da API, identificando possíveis pontos de melhoria e assegurando a manutenibilidade do projeto. A ferramenta JaCoCo foi utilizada para garantir a cobertura mínima de testes e oferecer insights sobre as áreas do código que necessitavam de atenção adicional.

O sistema de testes foi projetado para recriar ambientes que simulassem o ambiente real de produção, permitindo uma avaliação mais precisa da aplicação em cenários representativos. Essa estratégia foi acompanhada pela organização do código utilizando os princípios SOLID E boas práticas de programação, que proporcionaram um sistema modular, flexível e de fácil manutenção. A aplicação dessas práticas não apenas elevou a qualidade do sistema atual, como também, preparou a base do projeto para evoluções futuras com menor esforço, garantindo um produto final sólido e sustentável.

Ao final do desenvolvimento, foram realizados testes de aceitação com os usuários do sistema. Esses testes visaram verificar se o sistema atende aos critérios de aceitação e está alinhado com as expectativas dos usuários finais. Para isso, após a hospedagem do sistema, ele foi liberado para que o usuário pudesse validá-lo. Após alguns dias de testes, os proprietários consideraram o sistema muito útil, especialmente pela fácil visualização dos dados dos imóveis bem como a projeção dos valores de aluguéis e despesas. Os usuários sugeriram pequenas alterações relacionadas ao layout e a possibilidade de atualização no valor do aluguel. Foi constatado durante esta etapa de testes, que havia alguns bugs quanto ao update nas datas dos contratos de aluguéis. Foram então realizados as correções dos bugs e documentadas as necessidades para implementações futuras.

10.1 MELHORIAS FUTURAS

Como próximo passo, planeja-se a implementação de um dashboard mais completo, integrando todos os endpoints da API que ainda não foram expostos no front-end bem como

a opção de atualizar o valor dos contratos de aluguel. Além disso, pretende-se desenvolver um módulo específico para manipulação de arquivos e imagens, permitindo a adição de fotos aos imóveis, registros de manutenção e documentos associados aos contratos de aluguel. Esses recursos poderão facilitar o gerenciamento visual dos imóveis e das manutenções, além de documentar todas as etapas do ciclo de aluguel, proporcionando uma experiência mais completa e organizada para os usuários, caso sejam considerados necessários.

A aplicação, com seu robusto sistema de login, poderá permitir que os locatários cadastrem pedidos de manutenção nos imóveis, além de possibilitar o controle de manutenções realizadas por usuários externos. Esses usuários podem registrar informações detalhadas sobre os serviços prestados, incluindo a inserção de fotos ao cadastrar despesas relacionadas a esse tipo de atividade.

É importante ressaltar, que o sistema carece da funcionalidade de inserção de documentos estáticos para documentação dos processos de aluguéis, bem como cadastro de imagens para monitoramento de estados dos imóveis e controle de manutenções, porém, tal funcionalidade requer implementação e uso de ferramentas mais complexas, que demandam um tempo maior para sua implementação para garantir a escalabilidade saudável da aplicação.

Como melhorias futuras de engenharia, a API poderá ser configurada para segregar os testes unitários, de integração e ponta a ponta em módulos específicos, facilitando a organização e a manutenção do código. Além disso, a integração do Sonar à pipeline permitirá que apenas os *merge requests* que atendam aos critérios de qualidade definidos sejam aprovados, assegurando maior robustez e confiabilidade ao sistema.

11 REFERÊNCIAS

ANT DESIGN. Introdução ao Ant Design com React. Disponível em: https://ant.design/docs/react/introduce. Acesso em: 15 nov. 2024.

BAELDUNG. Difference Between JVM, JRE, and JDK, 2019. Disponível em: https://www.baeldung.com/jvm-vs-jre-vs-jdk. Acesso em: 04 abr. 2024. (BAELDUNG, 2019)

BAELDUNG. Java Classes and Objects, 2020. Disponível em: https://www.baeldung.com/java-classes-objects. Acesso em: 04 jun. 2024. (BAELDUNG, 2020)

BAELDUNG. Stack Memory and Heap Space in Java, 2022. Disponível em: https://www.baeldung.com/java-stack-heap. Acesso em: 04 jun. 2024. (BAELDUNG, 2022)

BABEL. Babel Documentation. Disponível em: https://babel.dev/docs. Acesso em: 27 out. 2024.

CHOPRA, Sunil; MEINDL, Peter. Gerenciamento da Cadeia de Suprimentos: estratégia, planejamento e operação. São Paulo: Prentice Hall, 2003. (CHOPRA, MEINDL, 2003)

CODD, E. F. A relational model of data for large shared data banks. Communications of the ACM, 26(1):64–69, 1983.

CORRÊA, Guilherme Augusto Machado de Almeida. A utilização de metodologias ágeis em projetos de desenvolvimento de software, 2017. Dissertação (Mestrado em Engenharia Elétrica) - Universidade de São Paulo, São Paulo. Disponível em: https://www.teses.usp.br/teses/disponiveis/3/3136/tde-11042017-143311/. Acesso em: 30 jun. 2024.

COUTO, Herderson; SILVA, Francisco Airton; CALLOU, Gustavo; ANDRADE, Ermeson. Uma Abordagem Experimental para Avaliar o Desempenho do Banco de Dados Open-Source PostgreSQL. ANAIS DA ESCOLA REGIONAL DE INFORMÁTICA DE GOIÁS (ERI-GO), 2022. Disponível em: https://sol.sbc.org.br/index.php/erigo/article/view/22533/22357. Acesso em: 04 jun. 2024. (COUTO, 2024)

DATE, C. J. Introdução a sistemas de bancos de dados, 2004. Elsevier Brasil. Disponível em: https://books.google.com.br/books?id=xBeO9LSIK7UC&printsec=frontcover&hl=pt-BR&source=gbs_ge_summary_r&cad=0#v=onepage&q&f=false. Acesso em: 10 jun. 2024.

DEV MEDIA. A história dos bancos de dados. Disponível em: https://www.devmedia.com.br/a-historia-dos-banco-de-dados/1678. Acesso em: 10 out. 2024.(DEV MEDIA, 2024a)

DEV MEDIA. *Como funcionam as aplicações web*. DevMedia, 2012. Disponível em: https://www.devmedia.com.br/como-funcionam-as-aplicacoes-web/25888. Acesso em: 20 nov. 2024. (DEV MEDIA, 2012)

DEV MEDIA. *Desenvolvimento Orientado por Comportamento (BDD)*. Disponível em: https://www.devmedia.com.br/desenvolvimento-orientado-por-comportamento-bdd/21127. Acesso em: 15 dez. 2024.(DEV MEDIA, 2024b)

FOWLER, Martin; HUMBLE, Jez. Continuous delivery: reliable software releases through build, test, and deployment automation. Addison-Wesley Professional, 2010.

GIL, A. C. *Métodos e Técnicas de Pesquisa Social*. 6. ed. São Paulo: Atlas, 2019. Disponível em: https://ayanrafael.com/wp-content/uploads/2011/08/gil-a-c-mc3a9todos-e-tc3a9cnicas-de-pesquisa-social.pdf. Acesso em: 20 nov. 2024.

GIT, Scott Chacon, Ben Straub. Começando-Sobre-Controle-de-Versão, 2014. Disponível em: https://git-scm.com/book/pt-br/v2/Come%C3%A7ando-Sobre-Controle-de-Vers%C3%A3o. Acesso em: 30 jun. 2024.

GITHUB. GitHub. Disponível em: http://github.com. Acesso em 26 nov. 2024.

GOOGLE CLOUD. What is a relational database? Disponível em: https://cloud.google.com/learn/what-is-a-relational-database?hl=pt-BR. Acesso em: 10 out. 2024.

GROUP SOFTWARE, Belo Horizonte, 2024. Disponível em: https://www.groupsoftware.com.br/administracao-de-imobiliarias/imobiliaria21/. Acesso em: 03 set. 2024. (IMOBILIARIA 21, 2024)

GROUP SOFTWARE, Belo Horizonte, 2024. Disponível em: https://www.groupsoftware.com.br/sobre-a-group-software/. (GROUP S., 2024)

HAYES, Khaleel. ERP Financial Management Functions. Disponível em: https://www.selecthub.com/enterprise-resource-planning/erp-financial-management-functions. Acesso em: 03 set. 2024.

HEROKU. Home Page. Disponível em: https://heroku.com. Acesso em: 01 out. 2024. (HEROKU, 2024)

IMOBZI, São Paulo, 2024. Disponível em: https://www.imobzi.com/planos/. Acesso em: 04 abr. 2024. (IMOBZI, 2024b)

IMOBZI, São Paulo, 2024. Disponível em: https://www.imobzi.com/sobre-nos/. Acesso em: 03 set. 2024. (IMOBZI, 2024a)

INTERNET ENGINEERING TASK FORCE. RFC 7519: JSON Web Token (JWT). Disponível em: https://datatracker.ietf.org/doc/html/rfc7519. Acesso em: 20 out. 2024. (7519, 2024)

JEST. *Documentação Oficial do Jest*. Disponível em: https://jestjs.io/pt-BR/. Acesso em: 20 nov. 2024.

JUNIT. Documentation, 2024. Disponível em: https://junit.org/junit5/docs/current/user-guide/#overview. Acesso em: 30 jun. 2024.

KANBAN UNIVERSITY. O Guia Oficial do Kanban, 2021. Disponível em: https://kanban.university/wp-content/uploads/2021/04/The-Official-Kanban-Guide_Portuguese_A4.pdf. Acesso em: 30 jun. 2024.

MANIFEST, Agile, 2001. Disponível em: https://agilemanifesto.org/. Acesso em: 30 jun. 2024.

MARTIN, Robert C., 2008. Clean Code: A Handbook of Agile Software Craftsmanship. English Edition. Capítulo 09.

MARTIN, Robert C., 2009. Código limpo: Habilidades práticas do Agile Software.

MEIRELLES, F. S. Pesquisa Anual d FGVcia 33ª Edição. São Paulo: FGV EAESP, 2022. Disponível em: https://eaesp.fgv.br/producao-intelectual/pesquisa-anual-uso-ti. Acesso em: 03 set. 2024.

MIRO. *Miro: Ferramenta de Colaboração Visual*. Disponível em: https://miro.com/. Acesso em: 20 nov. 2024.

MOCKITO. Documentation, 2024. Disponível em: https://site.mockito.org/. Acesso em: 30 jun. 2024.

ORACLE. Java Documentation, 2024. Disponível em: https://www.oracle.com/java/. Acesso em: 04 abr. 2024. (ORACLE, 2024)

REACT, Documentation, 2024. Components and Props. Disponível em: https://legacy.reactjs.org/docs/components-and-props.html. Acesso em: 04 abr. 2024. (REACT, 2024b)

REACT, Documentation, 2024. Disponível em: https://legacy.reactjs.org/. Acesso em: 04 abr. 2024. (REACT, 2024a)

REACT, Documentation, 2024. Disponível em: https://legacy.reactjs.org/docs/testing.html#testing-components. Acesso em: 30 abr. 2024. (REACT, 2024c)

SELENIUM. Documentation, 2024. Disponível em: https://www.selenium.dev/. Acesso em: 30 jun. 2024.

SEMVER. *Semantic Versioning 2.0.0*. Disponível em: https://semver.org/. Acesso em: 15 dez. 2024.

SOUZA, Jefferson Paulo de; FIGUEIREDO, Ramon. *Sistema de Gestão de Imóveis*. Trabalho de Conclusão de Curso – Centro Universitário de Brasília, Brasília, 2014. Disponível em: https://repositorio.uniceub.br/jspui/bitstream/235/6553/1/21360331.pdf. Acesso em: 09 dez. 2024.

SOUZA, Anderson Fuhr. Spring Security. Disponível em: https://www.linkedin.com/pulse/spring-security-anderson-fuhr-souza-1bsxf/?trackingId=hzeTjLlGQHqzCbl8X9fS6g%3D%3D. Acesso em: 19 out. 2024.

SPRING SECURITY. Spring Security Reference Documentation. Disponível em: https://docs.spring.io/spring-security/reference/index.html. Acesso em: 19 out. 2024.

SPRING. Documentation, 2024. Why Spring. Disponível em: https://spring.io/why-spring. Acesso em: 30 jun. 2024.

SPRINGDOC. Introduction. Disponível em: https://springdoc.org/#Introduction. Acesso em: 19 out. 2024. (SPRINGDOC, 2024)

SOUZA, Anderson Fuhr. *Diagrama de Classes do Projeto Lar Certo Imóveis*. 2024. Disponível em: https://drive.google.com/file/d/1RIt3OtRytrgx2ItjZ7Ruf7pDqr4ff3tH/view?usp=sharing. Acesso em: 27 nov. 2024.

SWAGGER. API Documentation. Disponível em: https://swagger.io/solutions/api-documentation/. Acesso em: 19 out. 2024. (SWAGGER, 2024a)

SWAGGER. Why you should create an API definition. Disponível em: https://swagger.io/blog/api-development/why-you-should-create-an-api-definition/. Acesso em: 19 out. 2024. (SWAGGER, 2024b)

TECIMOB, Santa Catarina, 2024. Disponível em: https://tecimob.com.br/modelos/. Acesso em: 09 jun. 2024. (TECIMOB, 2024)

TANENBAUM, A. S.; WETHERALL, D. J. *Redes de Computadores*. 5. ed. São Paulo: Pearson, 2011.

TECIMOB. Disponível em: https://tecimob.com.br/. Acesso em: 09 jun. 2024.

TOTVS. ERP Protheus, São Paulo, 2024. Disponível em: https://www.totvs.com/sistema-de-gestao/?utm_campaign=s-institucional-erp&utm_source=google-search&utm_medium=cpc&utm_term=totvs-protheus%3Fgclid. Acesso em: 09 jun. 2024. (TOTVS, 2024b)

TOTVS. Quem somos, São Paulo, 2024. Disponível em: https://www.totvs.com/sobre/. Acesso em: 09 jun. 2024. (TOTVS, 2024a)

Apêndices

APÊNDICE A – PADRÃO DAS HISTÓRIAS DE USUÁRIO, CRITÉRIOS DE ACEITAÇÃO E BDD

Este apêndice apresenta o padrão utilizado para a criação das histórias de usuário, bem como as estruturas utilizadas para definir os critérios de aceitação e documentar os testes com base no BDD (Behavior-Driven Development). Esses elementos guiaram o desenvolvimento do sistema, garantindo alinhamento com os requisitos do projeto.

Os conceitos de BDD descritos neste apêndice estão baseados em DEV MEDIA (2024b).

Padrão das Histórias de Usuário

As histórias de usuário foram escritas seguindo o formato:

- **Como [quem]**,
- **Quero [o que]**,
- **Para [por quê]**.

Esse padrão visa descrever claramente os objetivos e necessidades dos usuários, alinhando os requisitos às funcionalidades propostas pelo sistema. Ele também facilita o entendimento por parte dos desenvolvedores e partes interessadas, mantendo o foco no valor entregue ao usuário.

Exemplo de história de usuário:

- **Como** gestor,
- **Quero** gerar relatórios financeiros mensais,
- **Para** analisar o desempenho da imobiliária e embasar decisões estratégicas.

Critérios de Aceitação

Para cada história de usuário, foram definidos critérios de aceitação que detalham as condições específicas para que uma funcionalidade seja considerada concluída com sucesso. Esses critérios seguem o formato:

- O sistema deve permitir [ação ou funcionalidade específica].
- Deve garantir [validação ou comportamento esperado].
- Deve exibir [mensagens ou resultados esperados].

Os critérios de aceitação asseguram que os requisitos de qualidade e funcionalidade sejam atendidos, além de oferecer uma base para os testes automatizados e manuais.

Estrutura de BDD (Behavior-Driven Development)

Os testes baseados em BDD foram documentados utilizando o formato “Dado, Quando, Então”. Essa abordagem permite descrever cenários de teste de forma clara, destacando as condições iniciais, as ações realizadas e os resultados esperados.

Formato dos cenários:

- **Dado** [condição inicial ou contexto],
- **Quando** [ação realizada],
- **Então** [resultado esperado].

Exemplo de cenário de teste:

- **Dado** que o administrador esteja autenticado no sistema,
- **Quando** ele acessar a funcionalidade de geração de relatórios,
- **Então** o sistema deve apresentar as opções de período e formato de exportação.

Nota complementar

As histórias de usuário, os critérios de aceitação e os cenários de BDD foram documentados em um repositório digital para facilitar o acesso e consulta. Esses documentos podem ser acessados por meio do seguinte link: https://drive.google.com/drive/folders/1PrLWu0nuKatZEZ6fjDe7J63cxCrEio9h.