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

GUSTAVO MARONEZI

SISTEMA WEB E MOBILE PARA AUXÍLIO NO ACOMPANHAMENTO DE PESSOAS EM ABRIGOS PÚBLICOS

TRABALHO DE CONCLUSÃO DE CURSO

PATO BRANCO
2020

GUSTAVO MARONEZI

SISTEMA WEB E MOBILE PARA AUXÍLIO NO ACOMPANHAMENTO DE PESSOAS EM ABRIGOS PÚBLICOS

Trabalho de Conclusão de Curso de graduação, apresentado à disciplina de Trabalho de Conclusão de Curso 2, do Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas, da Universidade Tecnológica Federal do Paraná, Câmpus Pato Branco, como requisito parcial para obtenção do título de Tecnólogo.

Orientador: Prof. Vinicius Pegorini

PATO BRANCO
2020

TERMO DE APROVAÇÃO

TRABALHO DE CONCLUSÃO DE CURSO

Sistema Web e Mobile para Auxílio no Acompanhamento de Pessoas em Abrigos Públicos

POR

GUSTAVO MARONEZI

Este trabalho de conclusão de curso foi apresentado no dia 16 de junho de 2020, como requisito parcial para obtenção do título de Tecnólogo em Análise e Desenvolvimento de Sistemas, pela Universidade Tecnológica Federal do Paraná. O acadêmico foi arguido pela Banca Examinadora composta pelos professores abaixo assinados. Após deliberação, a Banca Examinadora considerou o trabalho aprovado.

Banca examinadora:

Prof. MSc Vinicius Pegorini	**Profa. MSc Andreia Scariot Beulke**	**Profa. Dra. Mariza Miola Dosciatti**
Professor orientador	Professora convidada	Professora convidada

Prof. Dr. Edilson Pontarolo	**Profa. Dra. Mariza Miola Dosciatti**
Coordenador do Curso de Tecnologia em Análise e Desenvolvimento de Sistemas	Responsável pela Atividade de Trabalho de Conclusão de Curso

Documento assinado eletronicamente por **MARIZA MIOLA DOSCIATTI**, **PROFESSOR ENS BASICO TECN TECNOLOGICO**, em 17/06/2020, às 15:26, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

Documento assinado eletronicamente por **ANDREIA SCARIOT BEULKE**, **PROFESSOR ENS BASICO TECN TECNOLOGICO**, em 17/06/2020, às 15:28, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

Documento assinado eletronicamente por **VINICIUS PEGORINI**, **PROFESSOR ENS BASICO TECN TECNOLOGICO**, em 17/06/2020, às 15:38, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

Documento assinado eletronicamente por **EDILSON PONTAROLO**, **COORDENADOR(A) DE CURSO/PROGRAMA**, em 17/06/2020, às 16:36, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

A autenticidade deste documento pode ser conferida no site https://sei.utfpr.edu.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0, informando o código verificador **1480904** e o código CRC **343C47A2**.

RESUMO

As pessoas que residem em abrigos públicos, costumeiramente conhecidos como asilos quando se referem aos idosos, necessitam de cuidados relacionados à saúde. É importante que eles sejam adequadamente medicados. É comum haver idosos que não mais tem condições de por si só cuidar da posologia dos seus medicamentos, eles podem esquecer da quantidade adequada, do horário ou até se o medicamento já foi ministrado. Nesses locais há cuidadores que realizam essas funções. Contudo, como, em geral, são até dezenas de idosos residentes e com interações medicamentosas diversificadas em tipo, periodicidade e quantidade, um sistema de informação pode auxiliar as pessoas que realizam a função de ministrar os medicamentos. Assim, tendo como base um asilo da cidade de Pato Branco, mas considerando que a solução proposta pode ser aplicada para qualquer local que se caracterize como de atendimento a pessoas que necessitem de cuidados de saúde, foi modelado um sistema para a gestão da interação medicamentosa de residentes de abrigos. O sistema foi desenvolvido para plataforma *web* e possui uma interface responsiva para uso em dispositivos móveis para melhorar a experiência de usabilidade do cuidador, enfermeira ou médico em registrar que determinado medicamento foi ministrado para um paciente. O sistema também permite o registro de alergias gerais e medicamentosas visando gerar alertas quando cadastrado que um determinado residente passará a utilizar um medicamento que possui como princípio ativo o produto ao qual o residente é alérgico. O registro de medicamento que cada residente está fazendo uso permite determinar a previsão de quantidade de medicamento para que não haja desabastecimento ou desperdício. Um registro histórico de pressão arterial, temperatura, glicemia e peso também serão oferecidos pelo sistema.

Palavras-chave: Aplicação Web. Rich Internet Application. Gestão de interação medicamentosa.

ABSTRACT

People living in public shelters, commonly known as nursing homes, when referring to the elderly, need health-related care. It is important that they are properly medicated. It is common to have elderly people who are no longer able to take care of the dosage of their medications on their own, they can forget the adequate amount, the schedule or even if the medication has already been administered. In these places there are caregivers who perform these functions. However, as, in general, there are even dozens of elderly residents and with drug interactions diversified in type, periodicity and quantity, a system can help people who perform the function of administering medicines. Thus, based on an asylum in the city of Pato Branco, but considering that the proposed solution can be applied to any location that is characterized as serving people in need and health care, a system for the management of drug interaction was modeled of shelter residents. The system was developed for a web platform and has a responsive interface for use on mobile devices to improve the usability experience of the caregiver, nurse or doctor in registering that a certain medication was administered to the referred patient. The system also allows the registration of general and drug allergies in order to generate alerts when it is registered that a certain resident will start to use a medication that has as active ingredient the product to which the resident is allergic. The medication registration that each resident is using allows determining the forecast quantity of medication so that there is no shortage or waste. A historical record of blood pressure, temperature, blood glucose and weight will also be provided by the system.

Keywords: Web application. Rich Internet Application. Drug interaction management.

LISTA DE FIGURAS

Figura 1 - Crescimento da população idosa no Brasil no período de 2007 a 2027 10
Figura 2 - Projeção da população brasileira por grupos etários 11
Figura 3 - Dados de idosos em abrigos públicos 12
Figura 4 - Diagrama de Casos de Uso 24
Figura 5 - Diagrama de entidades e relacionamentos do banco de dados 28
Figura 6 - Tela de cadastro de usuários 37
Figura 7 - Tela de cadastro de usuários em dispositivos móveis 38
Figura 8 - Validação dos campos no cadastro de usuários 38
Figura 9 - Tela de autenticação do sistema 39
Figura 10 - Tela de autenticação em dispositivos móveis 39
Figura 11 - Tela de listagem de residentes 40
Figura 12 - Dashboard do Paciente 41
Figura 13 - Dashboard em dispositivos móveis 41
Figura 14 - Modal para inclusão de novo medicamento ao residente 42
Figura 15 - Listagem as avaliações de peso do residente 43
Figura 16 - Listagem das doenças do residente 43
Figura 17 - Filtros do relatório de previsão de medicamentos 44
Figura 18 - Relatório de Previsão de Medicamentos 45
Figura 19 - Organização do servidor 46
Figura 20 - Estrutura do cliente 56
Figura 21 - Listagem das funcionalidades 57

LISTA DE QUADROS

Quadro 1 - Lista de ferramentas e tecnologias 17
Quadro 2 - Requisitos funcionais 22
Quadro 3 - Requisitos não funcionais 23
Quadro 4 - Manter usuários 25
Quadro 5 - Operação de inclusão nos casos de uso “manter” e “registrar” 25
Quadro 6 - Operação alterar nos casos de “manter” e “registrar” 26
Quadro 7 - Operação realizar previsão de medicamentos 27
Quadro 8 - Campos da tabela Residentes 29
Quadro 9 - Campos da tabela Medicamentos 29
Quadro 10 - Campos da tabela ResidentesMedicamentos 30
Quadro 11 - Campos da tabela PrincipioAtivo 30
Quadro 12 - Campos da tabela ResidentesAlergiasGerais 30
Quadro 13 - Campos da tabela Medicamento_PrincipioAtivo 31
Quadro 14 - Campos da tabela ResidentesAlergiasMedic 31
Quadro 15 - Campos da tabela Cid 31
Quadro 16 - Campos da tabela ResidentesDoencas 32
Quadro 17 - Campos da tabela Funcoes 32
Quadro 18 - Campos da tabela Usuario 32
Quadro 19 - Campos da tabela Voluntarios 33
Quadro 20 - Campos da tabela Peso 34
Quadro 21 - Campos da tabela AvaliacaoPressaoArterial 34
Quadro 22 - Campos da tabela AvaliacaoTemperatura 35
Quadro 23 - Campos da tabela HistoricosClinicos 35
Quadro 24 - Campos da tabela UnidadeMedicamento 35
Quadro 25- Campos da tabela Cidade 35
Quadro 26 - Campos da tabela Papel 36
Quadro 27 - Campos da tabela Usuario_Papeis 36
Quadro 28 - Campos da tabela Residente_Glicemia 36

LISTAGENS DE CÓDIGOS

Listagem 1 – *Controller* de autenticações 47
Listagem 2 - Método *generateToken()* 48
Listagem 3 - Classe *AuthenticationTokenFilter* 49
Listagem 4 - Classe WebSecurity 50
Listagem 5 - Classe *CrudController* 52
Listagem 6 - Método para salvar usuários. 53
Listagem 7 - Métodos para salvar e validar usuários no *UsuarioServiceImpl* 54
Listagem 8 - Listagem do módulo principal 57
Listagem 9 - Arquivo *app-routing.module.ts* 59
Listagem 10 - Código *HTML* da página de login 60
Listagem 11 - Classe *components* do *login* 61
Listagem 12 - Classe *LoginService* 62
Listagem 13 - *Component* do menu 64
Listagem 14 - Classe em que é montado o HTML dos menus 66
Listagem 15 - *Service* de CRUD genérico 66
Listagem 16 - *ResidenteDoencaComponent* 67
Listagem 17 - Evento de click ao clicar no botão Salvar 69
Listagem 18 - *Table* que exibe as doenças dos residentes 69

LISTA DE SIGLAS

CID	Classificação Estatística Internacional de Doenças Relacionados Com a Saúde
HTML	*Hypertext Markup Language*
HTTP	*Hypertext Transfer Protocol*
IBGE	Instituto Brasileiro de Geografia e Estatística
IMC	Índice de Massa Corporal
JWT	*Json Web Token*
PNAD	Pesquisa Nacional por Amostra de Domicílios
RF	Requisito Funcional
RIA	*Rich Internet Applications*
RNF	Requisito Não Funcional
URL	*Uniform Resource Locator*

SUMÁRIO

1 INTRODUÇÃO 10
1.1 CONSIDERAÇÕES INICIAIS 10
1.2 OBJETIVOS 13
1.2.1 OBJETIVO GERAL 13
1.2.2 OBJETIVOS ESPECÍFICOS 13
1.3 JUSTIFICATIVA 13
1.4 ESTRUTURA DO TRABALHO 14
2 APLICAÇÕES INTERNET RICAS 15
3 MATERIAIS E MÉTODO 17
3.1 MATERIAIS 17
3.2 MÉTODO 18
3.2.1 LEVANTAMENTO DE REQUISITOS 18
3.2.2 ANÁLISE E PROJETO 19
3.2.3 IMPLEMENTAÇÃO 19
3.2.4 TESTES 20
4 RESULTADOS 21
4.1 ESCOPO DO SISTEMA 21
4.2 MODELAGEM DO SISTEMA 21
4.3 APRESENTAÇÃO DO SISTEMA 37
4.4 IMPLEMENTAÇÃO DO SISTEMA 45
4.4.1 SERVIDOR 45
4.4.2 CLIENTE 55
5 CONCLUSÃO 71
REFERÊNCIAS 72

1 INTRODUÇÃO

Este capítulo apresenta as considerações iniciais que definem o contexto no qual se insere a aplicação desenvolvida como resultado da realização deste trabalho. Neste capítulo também são apresentados os objetivos e a justificativa do trabalho. E, por fim, está a organização do texto por meio da apresentação dos seus capítulos subsequentes.

1.1 CONSIDERAÇÕES INICIAIS

Em 2007 a população idosa (pessoas com mais de 59 anos) no Brasil era de 17 milhões (DINO, 2019). Dados indicam que a população brasileira com 60 anos ou mais vem mantendo a tendência de envelhecimento sendo verificada nos últimos anos, passando para 26 milhões em 2017. Esses dados são provenientes da Pesquisa Nacional por Amostra de Domicílios Contínua (PNAD) – Características dos Moradores e Domicílios, realizada pelo Instituto Brasileiro de Geografia e Estatística (IBGE) (PARADELLA, 2017). E projeções do IBGE apontam que em 2027 essa parcela da população será de 37 milhões (DINO, 2019).

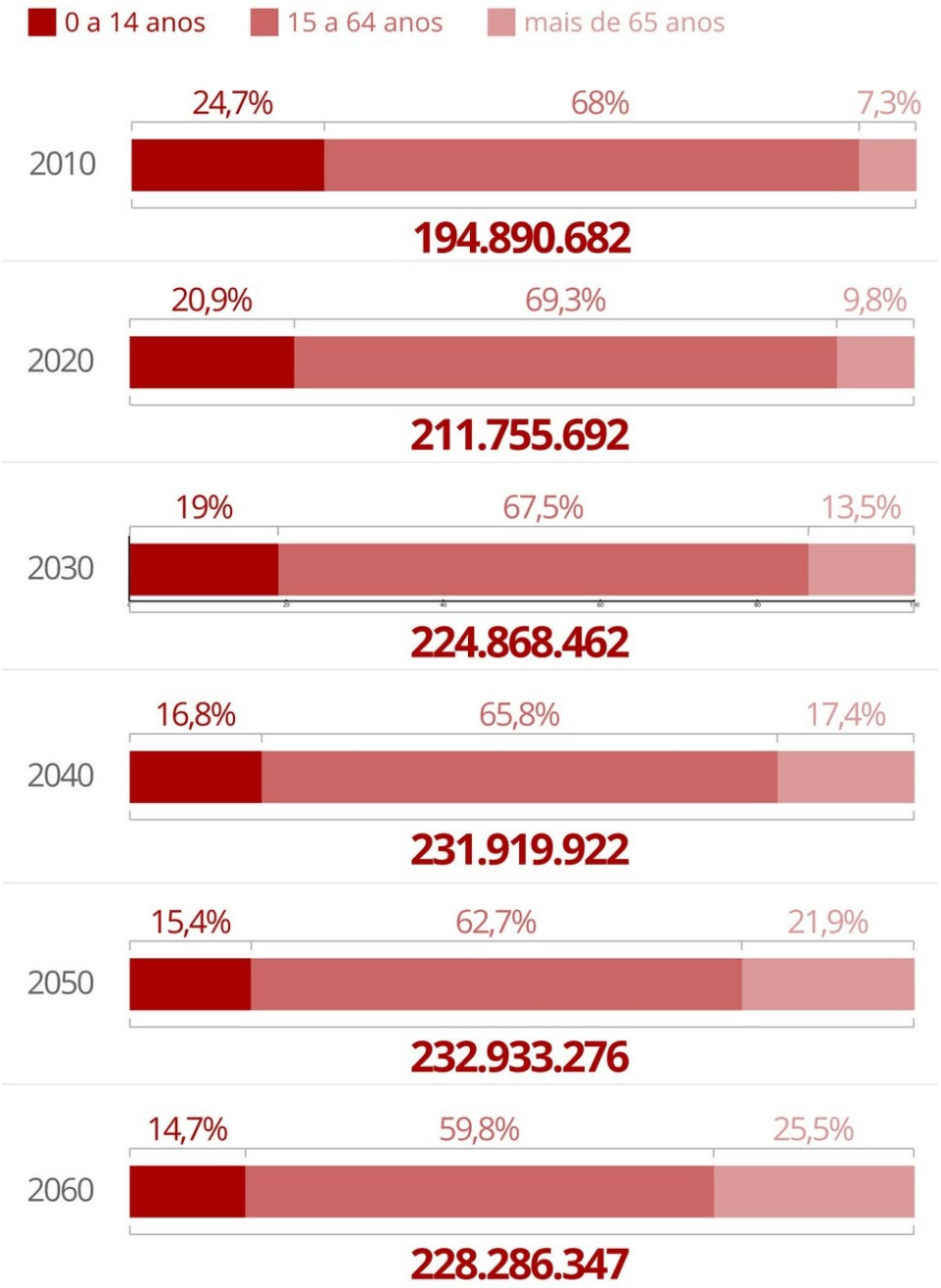
A Figura 1 apresenta um gráfico que mostra o crescimento da população idosa no Brasil no período de 2007 a 2017 e o crescimento projetado para 2027.

Figura 1 - Crescimento da população idosa no Brasil no período de 2007 a 2027

Fonte: DINO (2019, p.s.n.).

A curva ascendente apresentada na Figura 1 indica que a população brasileira está em trajetória de envelhecimento e o percentual de pessoas com mais de 65 anos passará dos atuais (em 2018) 9,2% para 25,5%. Essa projeção foi realizada pelo IBGE (ALVARENGA; BRITO, 2018). A Figura 2 apresenta a projeção para a população brasileira para pelos grupos etários: de 0 a 14 anos, de 15 a 64 e com mais de 64 anos.

Figura 2 - Projeção da população brasileira por grupos etários

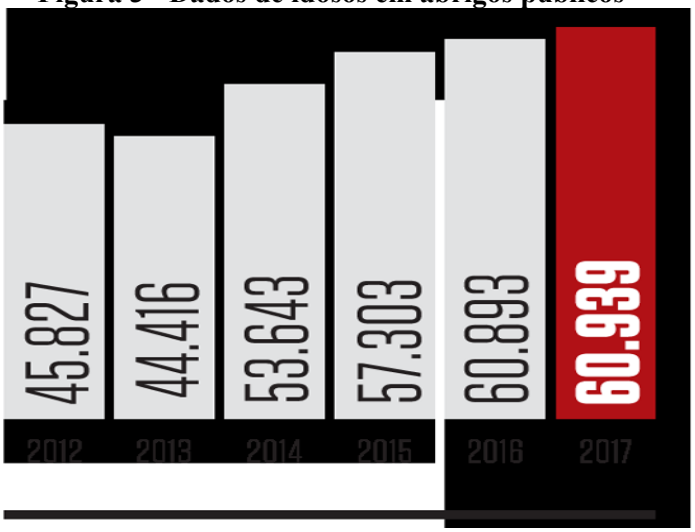


Fonte: IBGE[1] (2018 *apud* Alvarenga e Brito, 2018).

[1]IBGE. **Projeção da população brasileira.** 2018. Disponível em: https://www.ibge.gov.br/estatisticas/sociais/populacao. Acesso em: 16 jun. 2020.

O envelhecimento da população - como mostram os dados da Figura 2 que apresenta que em 2060 mais de 25% da população será composta por idosos - pode trazer problemas decorrentes da falta de planejamento que podem ser de difícil solução. Entre esses problemas pode ser citado o crescimento do número de pessoas em asilos e a falta de recursos financeiros próprios que podem assegurar uma velhice mais tranquila ou pelo menos com as condições de atendimento à saúde e aos cuidados necessários às pessoas idosas. No Brasil, a população de idosos em alojamentos públicos cresceu 33% em cinco anos (VILARDAGA; CAVICCHIOLI, 2018). A Figura 3 apresenta dados de idosos que estão em abrigos públicos. Esses dados indicam o crescimento dessa população.

Figura 3 - Dados de idosos em abrigos públicos



Fonte: Vilardaga e Cavicchioli (2018, p.s.n.).

As estatísticas mostram que o brasileiro não se prepara adequadamente para enfrentar o momento em que sua força produtiva se esgota e que ele depende unicamente dos recursos financeiros provenientes da aposentadoria. Um levantamento realizado pelo Banco Mundial revela que a formação de uma poupança privada no Brasil para sustentar os idosos do futuro não é adequada. Em um *ranking* de mais de 144 países, o Brasil ocupa o 101° lugar em termos de reserva de aposentadoria. Em 2017, apenas 11% dos brasileiros declararam poupar para a velhice. No Canadá, por exemplo, esse percentual é de 59% (VILARDAGA; CAVICCHIOLI, 2018).

Os dados de quantidade de idosos em abrigos públicos e a natural necessidade de assistência médica, com medicação de uso contínuo, para os idosos, sustentam a necessidade do auxílio de sistemas e aplicativos computacionais para o controle do histórico de saúde

dessas pessoas e da ingestão de medicamentos. É nesse contexto e escopo que está a proposta do sistema a ser desenvolvido como resultado deste trabalho.

1.2 OBJETIVOS

O objetivo geral apresenta o resultado principal pretendido com a implementação do sistema que é objeto desta proposta. Os objetivos específicos destacam funcionalidades e aplicabilidade do sistema desenvolvimento, complementando o objetivo geral.

1.2.1 OBJETIVO GERAL

Desenvolver um sistema *web* para registro e acompanhamento de dados relacionados à saúde e às interações medicamentosas de pessoas em abrigos públicos (asilos).

1.2.2 OBJETIVOS ESPECÍFICOS

- Auxiliar os cuidadores em asilos no controle dos medicamentos que são de uso contínuo ou periódico por idosos.
- Facilitar a gestão dos medicamentos ministrados pelo controle de qual medicamento, horário e período deve ser ministrado para cada idoso.
- Manter um registro simples do acompanhamento de histórico médico de cada idoso.

1.3 JUSTIFICATIVA

Uma forma de auxílio no controle das interações medicamentosas dos idosos é importante em decorrência dos remédios de uso contínuo e de acompanhamento de níveis de insulina, por exemplo. Em um abrigo público, como os asilos e as casas de apoio, em geral, são muitos os idosos que necessitam de acompanhamento quanto à medicação. Muitas vezes, esses idosos não têm condições, em decorrência de problemas de memória que podem acometê-los ou de problemas físicos, de saber quais medicamentos precisam ingerir e o horário para isso, por exemplo.

Nesses ambientes há cuidadores e é comum haver voluntários que se encarregam de ministrar as medicações necessárias. Contudo, se houvesse o auxílio de um sistema

computacional, as informações dos residentes poderiam ser armazenadas de uma forma mais adequada e segura. Justifica-se, assim, a proposta deste trabalho que é o desenvolvimento de um aplicativo para armazenar dados pessoais e históricos de saúde dos idosos e, principalmente, das medicações que eles devem ingerir e do controle de posologia desses medicamentos.

1.4 ESTRUTURA DO TRABALHO

Este trabalho está organizado em capítulos. Este é o primeiro capítulo e apresenta as considerações iniciais com o contexto do sistema desenvolvido, os seus objetivos e a justificativa. O Capítulo 2 apresenta o referencial teórico sobre aplicações Internet com interface rica. No Capítulo 3 estão as ferramentas e as tecnologias utilizadas na modelagem do sistema e que foram utilizadas na implementação do sistema. No Capítulo 4 é apresentado o resultado da realização do trabalho que é a modelagem e o desenvolvimento do sistema. Por fim, estão as referências utilizadas no texto.

2 APLICAÇÕES INTERNET RICAS

Durante muitos anos, as aplicações *web* moveram-se das aplicações *web* tradicionais, as *webs* 1.0 nas quais a maior parte do processamento de informação era realizada no servidor enquanto o cliente tinha a responsabilidade de mostrar conteúdo estático, para as atuais aplicações Internet ricas (BERNARDI; DI LUCCA; DISTANTE, 2014). Aplicações *web* tradicionais possuem grandes limitações em termos de usabilidade e interatividade da sua interface com o usuário baseada em *Hypertext Markup Language* (HTML). Aplicações Internet ricas, as *Rich Internet Applications* (RIAs), surgiram visando superar essas limitações (BUSCH; KOCH, 2009).

O termo *Rich Internet Applications* foi introduzido em março de 2002 pela Macromedia para lidar com as limitações existentes de riqueza da interface das aplicações, em termos de mídia e conteúdo e visando proporcionar sofisticação das aplicações (ALLAIRE, 2002). Meliá *et al*. (2008) endossam essas colocações quando se referem que as aplicações *web* da época possuíam grandes limitações na usabilidade e interatividade de suas interfaces. As RIAs vieram como uma proposta para lidar com essas limitações, proporcionando interfaces com componentes gráficos mais ricos e mais eficientes, similares às aplicações *desktop*. Exemplos de características das aplicações *desktop* incorporadas às aplicações *web* que auxiliaram a caracterizá-las como ricas são: arrastar e soltar, uso de teclas de atalho, uso da capacidade computacional local (no cliente) e *feedback* imediato para os usuários quando eles interagem com a aplicação (BUSCH; KOCH, 2009).

Além dos aspectos de interface que costumeiramente caracterizam as RIAs, o lado cliente da aplicação realiza processamento complexo de dados, minimizando a transferência de dados entre cliente e servidor, movendo as camadas de interação e apresentação do servidor para o cliente. Nessa arquitetura, uma RIA é carregada pelo cliente junto com alguns dados iniciais, em seguida o cliente gerencia os dados, processando a renderização da página e realizando o gerenciamento de eventos, comunicando-se com o servidor quando o usuário requer mais informações ou enviar dados (BOZZON *et al*., 2006). A arquitetura das RIAs também é caracterizada pelo assincronismo entre as requisições do cliente e a resposta do servidor para essas requisições (MACHADO; BERNARDO FILHO; RIBEIRO, 2009).

As RIAs são caracterizadas por uma interação mais ampla e melhorada com o usuário que não possui somente um papel passivo (isto é, somente como um consumidor de conteúdo armazenado e processado no lado servidor da aplicação), mas tendo assumido um papel ativo

baseado nas escolhas realizadas pelo usuário para a definição ou processamento do conteúdo a ser fornecido para o lado cliente da aplicação (BERNARDI; DI LUCCA; DISTANTE, 2014).

Esse tipo de aplicação oferece uma série de vantagens, elas provem interface gráfica mais rica e mais eficiente assemelhando-se às aplicações *desktop*. Contudo, elas são aplicações *web* complexas baseadas em uma arquitetura cliente mais robusta, comunicação assíncrona e uma grande variedade de recursos de interface. O cliente pode gerenciar dados e processos e o tráfego de rede é reduzido pela comunicação melhorada. Essas características melhoram a usabilidade e ampliam as possibilidades de interação dos usuários (BUSCH; KOCH, 2009).

Além de as RIAs apresentarem melhor usabilidade, elas possibilitam melhor desempenho porque elas reduzem significativamente a quantidade de requisições para o servidor pelo processamento no cliente de muitas das operações realizadas e pela manipulação de dados solicitados pelos usuários, reduzindo, assim, o recarregamento das páginas e o tráfego na Internet, que ocorre entre cliente e servidor (BERNARDI; DI LUCCA; DISTANTE, 2014).

As RIAs são, de fato, baseadas em um novo estilo arquitetural cliente-servidor que usa requisições assíncronas compostas de pequenos blocos de dados. As RIAs são similares às aplicações *desktop* porque elas combinam a arquitetura de distribuição com a interatividade de interface e o poder computacional das aplicações *desktop* e a combinação resultante melhora todos os elementos de uma aplicação *web* (dados, lógica de negócio, comunicação e apresentação (FRATERNALI; ROSSI; SANCHEZ-FIGUEROA, 2010). Resumidamente, uma RIA pode ser vista como uma aplicação *web* na qual a interface com o usuário é processada no lado cliente e a lógica de negócio é definida por serviços *backend* (VALVERDE; PASTOR, 2009).

3 MATERIAIS E MÉTODO

A seguir estão os materiais e o método utilizados para a modelagem e a implementação do sistema obtido como resultado deste trabalho.

3.1 MATERIAIS

O Quadro 1 apresenta a lista de ferramentas e tecnologias que foram utilizadas para o desenvolvimento deste trabalho.

Quadro 1 - Lista de ferramentas e tecnologias

Ferramenta / Tecnologia	Versão	Disponível em:	Finalidade
HTML	5	https://www.w3.org/html/	Linguagem de marcação de hipertexto
CSS	4	https://www.w3.org/Style/CSS	Linguagem de folhas de estilo em cascata
TypeScript	3.5.3	https://www.typescriptlang.org	Linguagem de programação
Angular	8.3.23	https://angular.io/	Framework para o lado cliente
PostgreSQL	10	https://www.postgresql.org/	Gerenciador de banco de dados
pgAdmin	4 v4.3	https://www.pgadmin.org/	Administrador de banco de dados
Visual Paradigm	15.2	https://www.visual-paradigm.com	Ferramenta de modelagem UML
npm	6.4.1	https://www.npmjs.com/	Package Manager para JavaScript
PrimeNG	8.0.0	https://www.primefaces.org/primeng/	Framework para o lado cliente
IntelliJ	2019.2.1	https://www.jetbrains.com/idea/	IDE para implementação de código
Visual Studio Code	2017	https://code.visualstudio.com/	IDE para implementação de código

Postman	7.0.7	https://www.getpostman.com	Rest client

Fonte: Autoria própria.

3.2 MÉTODO

O método consiste nas atividades de levantamento e modelagem de requisitos e na implementação do aplicativo. A seguir é descrita, sucintamente, a realização dessas etapas.

3.2.1 LEVANTAMENTO DE REQUISITOS

A ideia para implementação do aplicativo surgiu visando facilitar o controle dos medicamentos e dados básicos voltados à saúde e ao bem-estar dos residentes no asilo da cidade de Pato Branco.

As funcionalidades foram desenvolvidas a partir de informações levantadas com o auxílio da professora Beatriz T. Borsoi, baseando-se no que funcionários e voluntários precisam para atender as principais necessidades dos residentes em termos de gestão dos aspectos relacionados à saúde. Durante a análise, verificou-se a necessidade de um controle de medicamentos na aplicação, indicando quais medicamentos o residente usa, em qual horário e em qual período de tempo, informando se é diário, semanal ou mensal. Pelo fato de o sistema permitir o registro de todos os medicamentos em uso, será possível realizar uma previsão de medicamentos em um determinado período de tempo. Facilitando a gestão do estoque e evitando desperdício.

Outra motivação e circunstância de análise para definição dos requisitos foram a necessidade de ter-se um controle de avaliações básica sobre a saúde de residentes em asilos, casas de repouso e afins como, por exemplo, pressão arterial, peso e temperatura. Dessa forma, pode-se obter um histórico de Índice de Massa Corporal (IMC) do residente ou mesmo verificar se ele costuma ter febre ou descontrole da pressão arterial constantemente.

Também se verificou a necessidade de um historial clínico do residente, no qual os responsáveis pelo monitoramento da saúde pudessem adicionar informações complementares. Um exemplo de uso desse historial, seria quando o médico realiza uma consulta com o residente. Durante essa consulta ou ao final dela, o médico pode adicionar informações relacionadas ao referido residente.

Outra necessidade levantada foi a de ter um cadastro de alergias medicamentosas e um cadastro de outras alergias. As alergias medicamentosas devem ter iteração com o controle de medicamentos. No momento em que o usuário adicionar um medicamento, o sistema verificará se alguma composição está cadastrada nas alergias do residente. Caso esteja cadastrada, deve-se gerar um alerta ao usuário, informando essa situação. Com relação às alergias gerais, elas servirão para identificar, por exemplo, alergias a alimentos do residente cadastrado. Além do cadastro de alergias, também haverá um cadastro das doenças que acometem cada residente.

3.2.2 ANÁLISE E PROJETO

A análise e o projeto consistiram na modelagem dos requisitos, que foram agrupados em funcionais e não funcionais. Os funcionais foram organizados em casos de uso, definindo os atores e seus papéis no sistema. O banco de dados modelado consiste em dezesseis tabelas para persistência das informações, sendo que a maioria delas são relacionadas com a tabela de residentes. As tabelas foram descritas de maneira que seja facilmente possível identificar o tipo de dado de cada campo, os campos obrigatórios e os relacionamentos entre as tabelas.

3.2.3 IMPLEMENTAÇÃO

A implementação da aplicação foi dividida nas seguintes etapas:

1) Criação do Model

Na segunda etapa, o foco foi o desenvolvimento das classes principais da aplicação como, por exemplo, a classe “Residente”, com todos os atributos e ligações com outras classes, sendo cada classe uma tabela do banco de dados.

2) Configuração do Hibernate para persistência das informações

A terceira etapa esteve voltada para suprir as necessidades de persistência e acesso aos dados da aplicação com a configuração do Hibernate, criando as relações de todas as tabelas no banco de dados.

3) Configuração do *Repository*

Nessa etapa foi adaptado o *Repository*, que contém os métodos para adicionar, atualizar, remover e recuperar dados das tabelas.

4) Configuração dos *Controllers*

Nesta etapa foram desenvolvidos os *controllers*, que contém toda a configuração de rota da aplicação e, recebem as requisições e tratam elas por meio dos serviços da aplicação.

5) Controle de autenticação

Nessa fase, foi realizado o controle de autenticação do usuário, validando o acesso do mesmo por meio de login e senha.

6) Desenvolvimento do *front-end*

Nesta etapa foi desenvolvida a parte cliente da aplicação.

3.2.4 TESTES

Os testes foram informais, sem um plano de teste definido e tiveram como objetivo verificar a existência de erros de codificação e o atendimento dos requisitos. Os testes foram realizados pelo próprio autor do trabalho e tiveram como base os requisitos definidos para o sistema.

4 RESULTADOS

Este capítulo apresenta o resultado da realização deste trabalho que é a modelagem e o desenvolvimento de um software para gestão de abrigos, casas de repouso e outros estabelecimentos do gênero, conhecidos como asilos.

4.1 ESCOPO DO SISTEMA

O sistema visa auxiliar na realização das atividades dos cuidadores, profissionais e voluntários em abrigos para idosos. A solução foi inicialmente planejada para locais que são comumente caracterizados como asilos, mas pode ser aplicada para a gestão de atividades de ambientes nos quais há pessoas que necessitam de atenção e cuidados em relação à saúde no tocante às interações medicamentosas, por exemplo.

O sistema é *web* e contém cadastro de pessoas que vivem no local (idosos, no caso do contexto planejado para o sistema) e funcionários, voluntários e demais profissionais que realizam atividades no local. Vinculado ao cadastro de idoso, há um registro de doenças e outros que acometem o referido idoso e as respectivas interações medicamentosas e restrições alimentares, por exemplo.

O controle dos medicamentos que cada pessoa deve receber, com a posologia, o horário de ingestão ou aplicação, o período de uso do medicamento e outros, será acompanhado por um histórico médico. O sistema contém uma interface para dispositivos móveis que permite o acompanhamento das medicações a serem realizadas em cada idoso. Assim, a pessoa responsável poderá acompanhar quais medicamentos devem ser ministrados, os horários e marcar que a referida medicação foi aplicada, facilitando, dessa forma, o controle das interações medicamentosas realizadas.

Considerando que o sistema permitirá o registro de todos os medicamentos em uso por cada um dos idosos, será possível realizar uma previsão de medicamentos necessários (consumo) por período.

4.2 MODELAGEM DO SISTEMA

O Quadro 2 apresenta a listagem dos requisitos funcionais identificados para o sistema. Nesse quadro a sigla RF significa Requisito Funcional.

Quadro 2 - Requisitos funcionais

Identificação	Nome	Descrição
RF01	Cadastro de residentes	Cadastrar os residentes da casa de abrigo para idosos.
RF02	Cadastro de voluntários	Cadastrar os voluntários que realizam algum tipo de atividade no abrigo. Voluntários são pessoas que realizam atividades específicas periódicas ou esporadicamente, mas que não possuem vínculo empregatício com o abrigo. No cadastro de cada voluntário deverá ser preenchido o tempo em que ele ajudará. Caso não tenha um tempo definido, poderá ser colocado tempo como indeterminado.
RF03	Cadastro de funcionários	Cadastrar os funcionários do local.
RF04	Cadastro de funções	Por padrão, o sistema já terá alguns tipos de usuários cadastrados como administrador, gestor, profissional de saúde e cuidador. O sistema permitirá o cadastro de outras funções, conforme desejar. A tela de cadastro de funções deverá ser um modal que será aberto dentro da tela de cadastro de usuários e da tela de cadastro de voluntários.
RF05	Cadastro de medicamentos	Tela na qual os usuários poderão cadastrar medicamentos.
RF06	Controle de medicamentos	Controle de medicamentos que cada residente deve receber.
RF07	Histórico clínico	O histórico clínico será uma linha do tempo na qual os responsáveis pelo cuidado do residente, como enfermeira, cuidador, médico e psicólogo, poderão inserir informações a respeito da saúde do referido residente.
RF08	Controle de temperatura corporal	Os responsáveis pelo cuidado do residente poderão manter registro da temperatura corporal dele no sistema. Podendo ter, dessa forma, um histórico específico de temperatura. No registro de temperatura, o sistema colocará por padrão a data e horário atual, que poderão ser alteradas pelo usuário.
RF09	Controle de pressão arterial	Os responsáveis pelo cuidado do residente poderão inserir registros sobre a pressão arterial do residente. No registro de pressão arterial, o sistema colocará por padrão a data e horário atual, que poderão ser alteradas pelo usuário.
RF10	Previsão de medicamentos	Será possível realizar uma previsão do consumo de medicamentos necessários para um residente baseado num determinado período, que será definido pelo usuário, baseando-se na quantidade diária que o residente consome desse medicamento. Também será possível prever para todos os residentes a quantidade de cada

		medicamento por um período de tempo determinado.
RF11	Registro de doenças	Juntamente com o cadastro de cada residente, haverá um registro das doenças que o acometem.
RF12	Registro de alergias medicamentosas	Assim como o registro de doenças, no cadastro dos residentes haverá um registro de alergias medicamentosas. Esse cadastro possuirá interação com o controle de medicamentos.
RF13	Controle de alergias medicamentosas	No controle de medicamentos, quando um usuário inserir um medicamento para o qual o residente possui alergia, o sistema exibirá uma mensagem, informando que o residente possui alergia ao medicamento que está seno inserido.
RF14	Cadastro de peso	No cadastro do residente, o usuário poderá informar o peso atual.
RF15	Registro de alergias em geral	No cadastro do residente, deverá haver, além do registro de alergias medicamentosas, um registro de alergias em geral como, por exemplo, alergias a alimentos, outros produtos, etc.

Fonte: Autoria própria.

O Quadro 3 apresenta os Requisitos Não Funcionais (RNF) identificados para o sistema. Os requisitos não funcionais são relacionados ao uso da aplicação referente a desempenho, usabilidade, segurança, infraestrutura, dentre outros.

Quadro 3 - Requisitos não funcionais

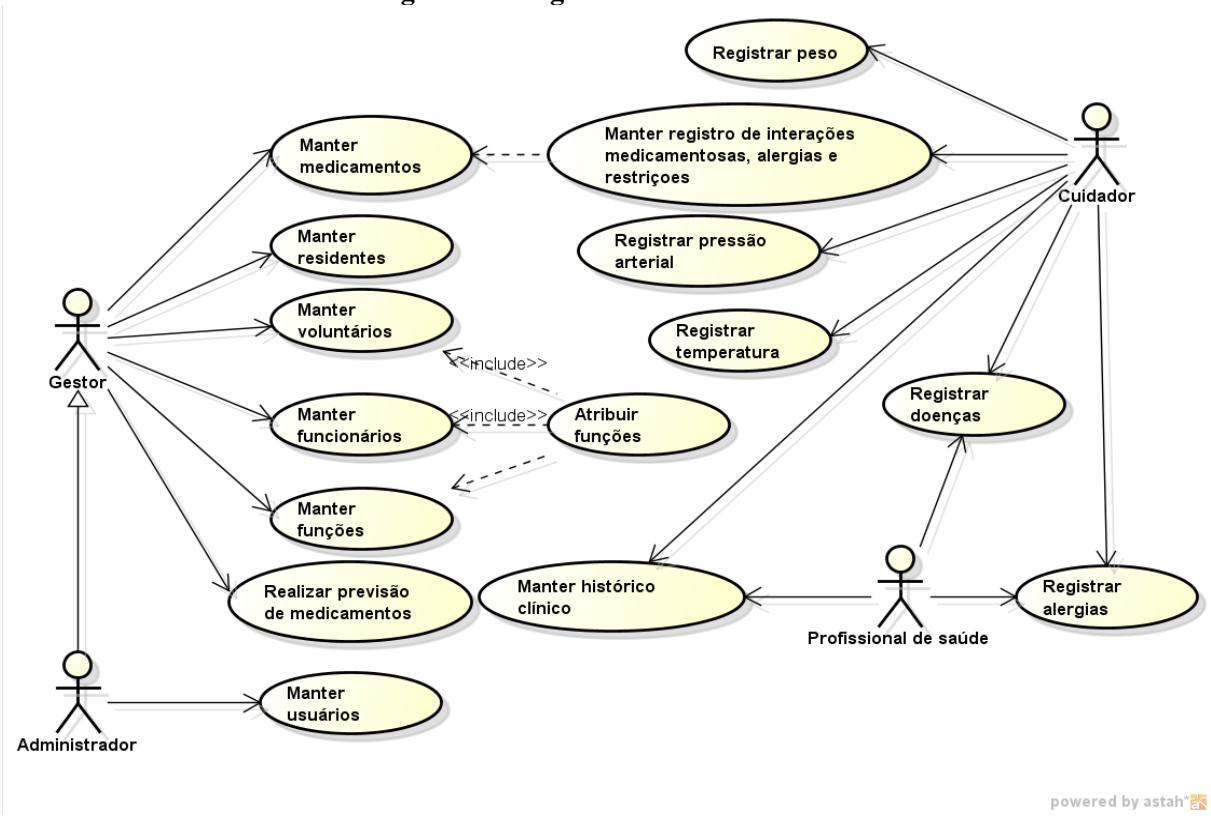
Identificação	Nome	Descrição
RNF01	Acesso ao sistema	O acesso ao sistema será realizado por meio de *login* e senha. Com permissões de acordo com o tipo de usuário.
RNF02	Uso de *design* responsivo nas interfaces gráficas	O sistema será construído para executar em ambiente *web* e deverá ter um leiaute responsivo, independente do *front-end* que será utilizado para acesso: navegador *web*, *smartphone* ou *tablet*.
RNF03	Compatibilidade com navegadores *web*.	O sistema deve funcionar nos principais navegadores como, por exemplo, Chrome, Firefox, Safari, Opera e Microsoft Edge.
RNF04	Controle de permissões	Cada usuário poderá utilizar apenas determinados módulos do software, com exceção do Administrador que realizará exclusivamente a manutenção de usuários no sistema e herdará as permissões do gestor (que é a manutenção dos cadastrados básicos do sistema). O administrador será o único usuário capaz de cadastrar outros usuários. Haverá o usuário do tipo Gestor que terá acesso a todos os módulos, com exceção do cadastro de usuários. Também terá o usuário

		Profissional de Saúde que poderão ser médicos, enfermeiras, psicólogas, dentro outros. Esse usuário terá acesso ao controle clínico do residente como, medições, alergias, histórico clínico e controle de medicamentos. E, ainda, o usuário Cuidador, que poderá registrar dados de temperatura, pressão e de administração de medicamentos aos residentes, conforme interação medicamentosa indicada para o referido residente.
RNF05	Cálculo de IMC	Na tela de cadastro de peso do residente, se o usuário desejar, poderá optar por realizar o cálculo de IMC. Para isso, deverá informar o peso em quilos, a altura em centímetros e, após isso, o sistema além de retornar o IMC, deverá informar se o residente está abaixo do peso, peso normal, sobrepeso, obesidade grau 1, 2 ou 3.

Fonte: Autoria própria.

O diagrama de casos de uso apresentado na Figura 4 contém as funcionalidades essenciais do sistema organizadas pelos seus atores que são: administrador, gestor, profissional de saúde e cuidador.

Figura 4 - Diagrama de Casos de Uso



Fonte: Autoria própria.

O Quadro 4 apresenta a expansão do caso de uso manter cadastro de usuário.

Quadro 4 - Manter usuários

<table>
<tr><td colspan="2">Caso de uso:
Manter cadastro de usuários
Descrição:
Esse caso de uso refere-se às operações de inclusão, exclusão, consulta e alteração do cadastro de usuários que terão acesso ao sistema.
Evento Iniciador:
Necessidade de incluir, excluir, editar, consultar ou alterar cadastros de usuários.
Atores:
Administrador.
Pré-condição:
Dados necessários disponíveis.
Sequência de Eventos:
1. Ator acessa o módulo ao qual deseja realizar a operação.
2. Sistema apresenta o formulário.
3. Ator realiza a operação desejada: incluir, excluir, consultar ou editar.
4. Sistema verifica se os dados para a operação estão consistentes e realiza a operação.
5. Sistema informa ao ator que os dados foram salvos.
Pós-Condição:
Dados do usuário inseridos no banco de dados.</td></tr>
<tr><td>Nome do fluxo alternativo (extensão)</td><td>Descrição</td></tr>
<tr><td>1. Dados não são válidos</td><td>1.1 No momento de salvar, o sistema faz a verificação e constata que há dados inválidos.
1.2 É emitida mensagem e retornado para a tela de edição.</td></tr>
<tr><td>2. Usuário inserido já existe.</td><td>2.1 No momento de salvar, o sistema faz a verificação e constata que o login cadastrado existe no sistema.
2.2 É emitida mensagem e retornado para a tela de edição.</td></tr>
</table>

Fonte: Autoria própria.

O Quadro 5 apresenta a expansão de inclusão dos casos de uso "manter" e "registrar".

Quadro 5 - Operação de inclusão nos casos de uso "manter" e "registrar"

Caso de uso: Incluir (refere-se à operação de inclusão nos casos de uso "manter" e "registrar"). **Descrição**: Ator insere dados no sistema. **Evento Iniciador**: Necessidade de incluir registros no sistema. **Atores**: Administrador, gestor, cuidador ou profissional da saúde, de acordo com suas funções, conforme foram definidas no caso de uso. **Pré-condição**: Estar autenticado no sistema. **Sequência de Eventos**: 1. Ator acessa o módulo ao qual deseja incluir dados. 2. Sistema apresenta o formulário.

3. Ator preenche os campos e clica em “Salvar” para persistir as informações. 4. O sistema verifica se os dados estão consistentes e realiza a operação. 5. Sistema informa ao ator que os dados foram salvos. **Pós-Condição**: Registros inseridos no banco de dados.

Nome do fluxo alternativo (extensão)	Descrição
1. Dados não são válidos	1.1 No momento de salvar, o sistema faz a verificação e constata que há dados inválidos. 1.2 É emitida mensagem avisando que os dados inseridos são inválidos, não persistindo as alterações no banco de dados. 1.3 Sistema retorna para a tela de inclusão, destacando os campos que estão com forma inválida.
2. Campos obrigatórios não preenchidos	2.1 Ator clica em “Salvar” sem ter preenchido os campos obrigatórios. 2.2 Sistema realiza a validação e retorna uma mensagem informando que alguns campos obrigatórios não foram preenchidos, não persistindo os dados no banco. 2.3 O sistema retorna para a tela de inclusão, destacando os campos obrigatórios não preenchidos.

Fonte: Autoria própria.

A operação alterar os casos de uso “manter” e “registrar” é apresentada no Quadro 6.

Quadro 6 - Operação alterar nos casos de “manter” e “registrar”

Caso de uso: Alterar (refere-se à operação de alterar nos casos de uso “manter” e “registrar”). **Descrição**: Ator edita dados no sistema. **Evento Iniciador**: Necessidade de editar registros no sistema. **Atores**: Administrador, gestor, cuidador ou profissional da saúde, de acordo com suas funções, conforme foram definidas no caso de uso. **Pré-condição**: Dados já devem ter sido incluídos no sistema. **Sequência de Eventos**: 1. Ator acessa a tela para visualizar os dados que já estão cadastrados. 2. Sistema apresenta o registro que foi selecionado para edição. 3. Ator altera os dados necessários e clica em “Salvar”. 4. O sistema verifica se os dados estão consistentes e realiza a operação. 5. Sistema informa ao ator que os dados foram salvos. **Pós-Condição**: Registros alterados no banco de dados.

Nome do fluxo alternativo (extensão)	Descrição
1. Dados não são válidos	1.1 No momento de salvar, o sistema faz a verificação e constata que há dados inválidos. 1.2 É emitida mensagem avisando que os dados inseridos são inválidos, não persistindo as alterações no banco de dados.

	1.3 Sistema retorna para a tela de edição, destacando os campos que estão com forma inválido.
2. Campos obrigatórios não preenchidos	2.1 Ator clica em "Salvar" sem ter preenchido os campos obrigatórios. 2.2 Sistema realiza a validação e retorna uma mensagem informando que alguns campos obrigatórios não foram preenchidos, não persistindo os dados no banco. 2.3 O sistema retorna para a tela de edição, destacando os campos obrigatórios não preenchidos.

Fonte: Autoria própria.

No Quadro 7 está a expansão do caso de uso realizar previsão de medicamentos.

Quadro 7 - Operação realizar previsão de medicamentos

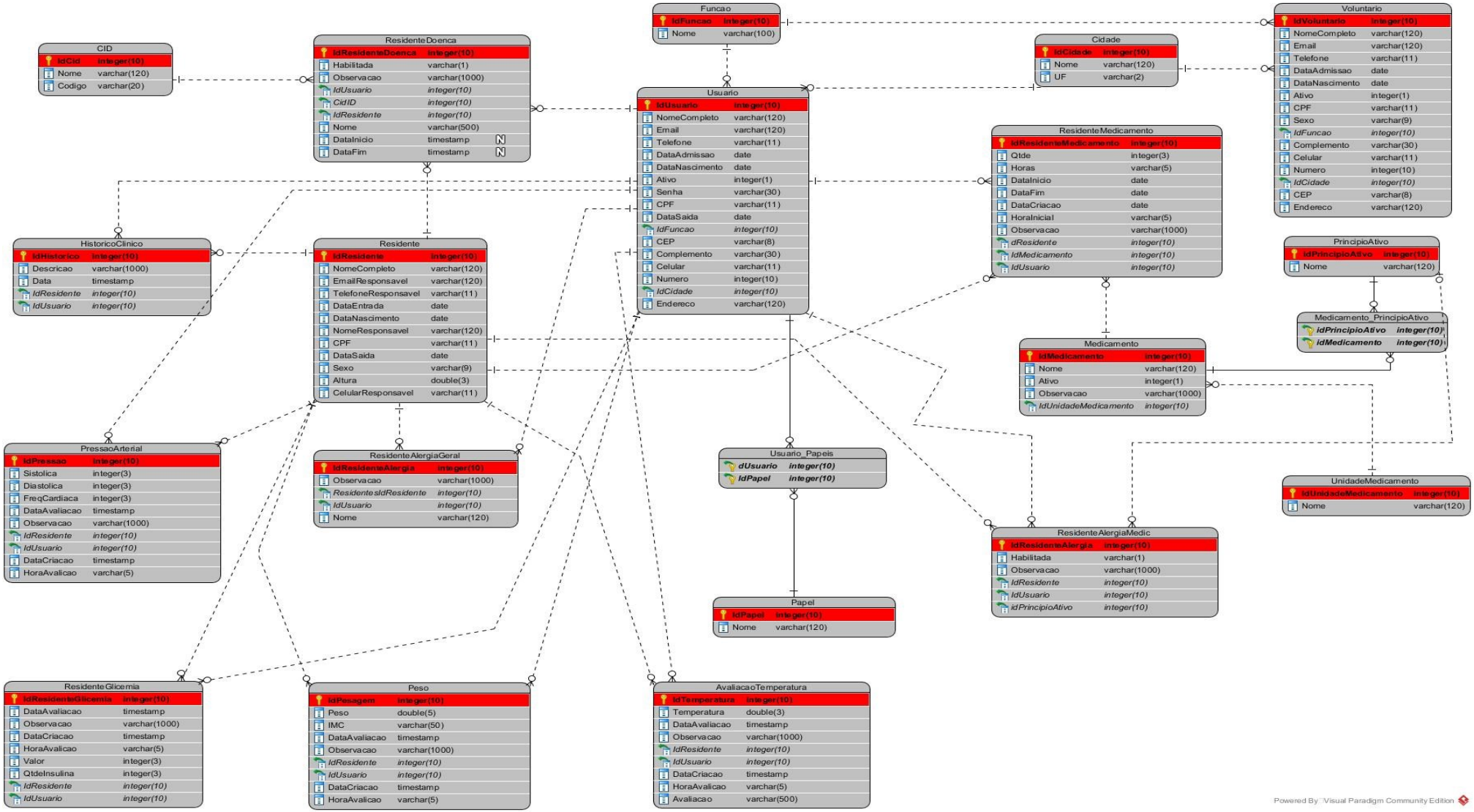
Caso de uso: Realizar previsão de medicamentos. **Descrição**: Ator exclui dados no sistema. **Evento Iniciador**: Necessidade de prever-se quanto de um determinado medicamento será utilizado num determinado período. **Atores**: Administrador ou gestor. **Pré-condição**: Estar autenticado no sistema. **Sequência de Eventos**: 1. Ator acessa a tela para emissão de relatório. 2. Ator realiza os filtros que desejar na tela. 3. Ator clica em "Gerar relatório". **Pós-Condição**: Relatório é gerado para o ator.

Nome do fluxo alternativo (extensão)	Descrição
1. Ator insere uma data inválida no filtro	1.1 Sistema exibe uma mensagem informando que as datas informadas são inválidas e retorna para a tela de geração do relatório.

Fonte: Autoria própria.

A Figura 5 apresenta o diagrama de entidades e relacionamentos, que representam o banco de dados da aplicação.

Figura 5 - Diagrama de entidades e relacionamentos do banco de dados



Fonte: Autoria própria.

A seguir está a descrição das tabelas do banco de dados apresentadas na Figura 5. As tabelas que possuem o campo idFuncionario, indicam o usuário do sistema que realizou o referido registro. O armazenamento desse campo será utilizado para identificar o responsável pelo registro.

No Quadro 8 estão os campos da tabela de Residente.

Quadro 8 - Campos da tabela Residente

Campo	Tipo	Nulo	Chave primária	Chave estrangeira	Observações
IdResidente	Numérico	Não	Sim	Não	
NomeCompleto	Texto	Não	Não	Não	
CPF	Texto	Não	Não	Não	
DataNascimento	Date	Não	Não	Não	
Sexo	Texto	Sim	Não	Não	
Altura	Numérico	Sim	Não	Não	
DataEntrada	Date	Não	Não	Não	
DataSaida	Date	Sim	Não	Não	
NomeResponsavel	Texto	Não	Não	Não	Pessoa de contato, responsável, familiar do/pelo residente
EmailResponsável	Texto	Sim	Não	Não	
TelelefoneResponsavel	Texto	Não	Não	Não	
CelularResponsavel	Texto	Não	Não	Não	

Fonte: Autoria própria.

Os campos da tabela Medicamento estão no Quadro 9.

Quadro 9 - Campos da tabela Medicamento

Campo	Tipo	Nulo	Chave primária	Chave estrangeira	Observações
IdMedicamento	Numérico	Não	Sim	Não	
Nome	Texto	Não	Não	Não	
Ativo	Numérico	Não	Não	Não	
Observação	Texto	Sim	Não	Não	
IdUnidadeMedicamento	Numérico	Não	Não	Sim	Vem da tabela UnidadeMedicamento

Fonte: Autoria própria.

No Quadro 10 estão os campos da tabela ResidenteMedicamentos. Nesta tabela serão armazenados todos os medicamentos que devem ser ministrados para cada residente.

Quadro 10 - Campos da tabela ResidentesMedicamentos

Campo	Tipo	Nulo	Chave primária	Chave estrangeira	Observações
IdResidenteMedicamento	Numérico	Não	Sim	Não	
IdResidente	Numérico	Não	Não	Sim	Da tabela Residente
IdMedicamento	Numérico	Não	Não	Sim	Da tabela Medicamento
Quantidade	Numérico	Não	Não	Não	
Periodicidade	Texto	Não	Não	Não	
DataInicio	Data	Não	Não	Não	
DataFim	Data	Não	Não	Não	
DataCriacao	Data	Não	Não	Não	
HoraInicio	Texto	Sim	Não	Não	
Observação	Texto	Sim	Não	Não	
IdUsuario	Numérico	Não	Não	Sim	Da tabela Usuario

Fonte: Autoria própria.

Os campos da tabela PrincipioAtivo são apresentados no Quadro 11.

Quadro 11 - Campos da tabela PrincipioAtivo

Campo	Tipo	Nulo	Chave primária	Chave estrangeira	Observações
IdPrincipioAtivo	Numérico	Não	Sim	Não	
Nome	Texto	Não	Não	Não	

Fonte: Autoria própria.

No Quadro 12 estão os campos da tabela ResidenteAlergiaGeral. Nesta tabela serão armazenadas todas as alergias gerais de cada residente. Nessa tabela não estão as alergias medicamentosas.

Quadro 12 - Campos da tabela ResidenteAlergiaGeral

Campo	Tipo	Nulo	Chave primária	Chave estrangeira	Observações
IdResidenteAlergia	Numérico	Não	Sim	Não	
IdResidente	Numérico	Não	Não	Sim	Campo que vem da tabela Residente
Nome	Texto	Não	Não	Não	
Observação	Texto	Sim	Não	Não	

IdUsuario	Numérico	Não	Não	Sim	Da tabela Usuario

Fonte: Autoria própria.

No Quadro 13 estão os campos da tabela Medicamento_PrincipioAtivo que armazena os princípios ativos que cada medicamento possui.

Quadro 13 - Campos da tabela Medicamento_PrincipioAtivo

Campo	Tipo	Nulo	Chave primária	Chave estrangeira	Observações
idPrincipioAtivo	Numérico	Não	Não	Sim	Campo que vem da tabela PrincipioAtivo
idMedicamento	Numérico	Não	Não	Sim	Campo que vem da tabela Medicamento

Fonte: Autoria própria.

No Quadro 14 estão os campos da tabela ResidenteAlergiasMedic. Nesta tabela serão armazenadas todas as alergias a medicamentos de determinados residentes.

Quadro 14 - Campos da tabela ResidenteAlergiasMedic

Campo	Tipo	Nulo	Chave primária	Chave estrangeira	Observações
IdResidenteAlergia	Numérico	Não	Sim	Não	
IdResidente	Numérico	Não	Não	Sim	Campo que vem da tabela Residente
IdPrincipioAtivo	Numérico	Não	Não	Sim	Campo que vem da tabela PrincipioAtivo
Habilitada	Numérico	Não	Não	Não	
Observação	Texto	Sim	Não	Não	
IdUsuario	Numérico	Não	Não	Sim	Da tabela Usuario

Fonte: Autoria própria.

No Quadro 15 estão os campos da tabela Cid.

Quadro 15 - Campos da tabela Cid

Campo	Tipo	Nulo	Chave primária	Chave estrangeira	Observações
IdDoenca	Numérico	Não	Sim	Não	
Nome	Texto	Não	Não	Não	
Codigo	Texto	Sim	Não	Não	

Fonte: Autoria própria.

No Quadro 16 estão os campos da tabela ResidenteDoenca.

Quadro 16 - Campos da tabela ResidenteDoenca

Campo	Tipo	Nulo	Chave primária	Chave estrangeira	Observações
IdResidenteDoenca	Numérico	Não	Sim	Não	
IdResidente	Numérico	Não	Não	Sim	Campo que vem da tabela Residente
IdCid	Numérico	Não	Não	Sim	Campo que vem da tabela Cid
Habilitada	Numérico	Não	Não	Não	
Observação	Texto	Sim	Não	Não	
Data	Data	Não	Não	Não	
IdUsuario	Numérico	Não	Não	Sim	Da tabela Usuario
Nome	Texto	Não	Não	Não	
DataInicio	Data	Sim	Não	Não	
DataFim	Data	Sim	Não	Não	

Fonte: Autoria própria.

No Quadro 17, estão os campos da tabela Funcoes, que se referem às funções dos funcionários e dos voluntários.

Quadro 17 - Campos da tabela Funcoes

Campo	Tipo	Nulo	Chave primária	Chave estrangeira	Observações
IdFuncao	Numérico	Não	Sim	Não	
Nome	Texto	Não	Não	Não	

Fonte: Autoria própria.

No Quadro 18 estão os campos da tabela Usuario, que manterá os usuários e funcionários do local.

Quadro 18 - Campos da tabela Usuario

Campo	Tipo	Nulo	Chave primária	Chave estrangeira	Observações
IdUsuario	Numérico	Não	Sim	Não	
IdFuncao	Numérico	Não	Não	Sim	Campo que vem da tabela Funcoes
NomeCompleto	Texto	Não	Não	Não	
Email	Texto	Sim	Não	Não	
Telefone	Texto	Sim	Não	Não	
CPF	Texto	Não	Não	Não	
DataAdmissao	Data	Sim	Não	Não	
DataNascimento	Data	Sim	Não	Não	

DataSaida	Data	Sim	Não	Não	
Senha	Texto	Sim	Não	Não	
Ativo	inteiro				Se possui acesso ao sistema
CEP	Texto	Não	Não	Não	
Complemento	Texto	Sim	Não	Não	
Celular	Texto	Não	Não	Não	
Numero	Numérico	Não	Não	Não	
Endereço	Texto	Não	Não	Não	
IdCidade	Numérico	Não	Não	Sim	Campo que vem da tabela Cidade

Fonte: Autoria própria.

No Quadro 19 estão os campos da tabela Voluntarios.

Quadro 19 - Campos da tabela Voluntarios

Campo	**Tipo**	**Nulo**	**Chave primária**	**Chave estrangeira**	**Observações**
IdVoluntario	Numérico	Não	Sim	Não	
IdFuncao	Numérico	Não	Não	Sim	Campo que vem da tabela Funcoes
NomeCompleto	Texto	Não	Não	Não	
Email	Texto	Sim	Não	Não	
Telefone	Texto	Sim	Não	Não	
CPF	Texto	Não	Não	Não	
Sexo	Texto	Sim	Não	Não	
DataEntrada	Data	Sim	Não	Não	
DataNascimento	Data	Sim	Não	Não	
DataSaida	Data	Sim	Não	Não	
Ativo	inteiro	Sim	Não	Não	
CEP	Texto	Não	Não	Não	
Complemento	Texto	Sim	Não	Não	
Celular	Texto	Não	Não	Não	
Numero	Numérico	Não	Não	Não	
Endereço	Texto	Não	Não	Não	
IdCidade	Numérico	Não	Não	Sim	Campo que vem da tabela Cidade

Fonte: Autoria própria.

No Quadro 20 estão os campos da tabela Peso, que se referem ao controle de peso de cada residente.

Quadro 20 - Campos da tabela Peso

Campo	Tipo	Nulo	Chave primária	Chave estrangeira	Observações
IdPeso	Numérico	Não	Sim	Não	
IdResidente	Numérico	Não	Não	Sim	Campo que vem da tabela Residente
Peso	Numérico	Não	Não	Não	
IMC	Decimal	Sim	Não	Não	
DataAvaliacao	Date	Não	Não	Não	
DataCriacao	Date	Não	Não	Não	
HoraAvaliacao	Texto	Não	Não	Não	
Observacao	Texto	Sim	Não	Não	
IdUsuario	Numérico	Não	Não	Sim	Da tabela Usuario

Fonte: Autoria própria.

No Quadro 21 estão os campos da tabela PressaoArterial, que armazenará todo o histórico da pressão arterial do residente.

Quadro 21 - Campos da tabela PressaoArterial

Campo	Tipo	Nulo	Chave primária	Chave estrangeira	Observações
IdPressao	Numérico	Não	Sim	Não	
IdResidente	Numérico	Não	Não	Sim	Campo que vem da tabela Residente
Sistolica	Numérico	Não	Não	Não	
Diastolica	Numérico	Não	Não	Não	
FreqCardiaca	Numérico	Sim	Não	Não	
DataAvaliacao	Date	Não	Não	Não	
Observacao	Texto	Sim	Não	Não	
IdUsuario	Numérico	Não	Não	Sim	Da tabela Usuario
DataCriacao	Date	Não	Não	Não	
HoraAvaliacao	Texto	Não	Não	Não	

Fonte: Autoria própria.

No Quadro 22 estão os campos da tabela AvaliacaoTemperatura, na qual ficará o histórico de avaliação de temperatura arterial do residente.

Quadro 22 - Campos da tabela AvaliacaoTemperatura

Campo	Tipo	Nulo	Chave primária	Chave estrangeira	Observações
IdTemperatura	Numérico	Não	Sim	Não	
IdResidente	Numérico	Não	Não	Sim	Campo que vem da tabela Residente
Temperatura	Numérico	Não	Não	Não	
Avaliação	Texto	Não	Não	Não	
DataAvaliacao	Date	Não	Não	Não	
HoraAvaliacao	Date	Não	Não	Não	
Observacao	Texto	Sim	Não	Não	
IdUsuario	Numérico	Não	Não	Sim	Da tabela Usuario
DataCriacao	Date	Não	Não	Não	

Fonte: Autoria própria.

No Quadro 23 estão os campos da tabela HistoricoClinicos.

Quadro 23 - Campos da tabela HistoricoClinico

Campo	Tipo	Nulo	Chave primária	Chave estrangeira	Observações
IdHistorico	Numérico	Não	Sim	Não	
idResidente	Numérico	Não	Não	Não	Da tabela Residente
Descricao	Texto	Não	Não	Não	
Data	Data	Não	Não	Não	
IdUsuario	Numérico	Não	Não	Sim	Da tabela Usuario

Fonte: Autoria própria.

No Quadro 24 estão os campos da tabela UnidadeMedicamento.

Quadro 24 - Campos da tabela UnidadeMedicamento

Campo	Tipo	Nulo	Chave primária	Chave estrangeira	Observações
IdUnidadeMedicamento	Numérico	Não	Sim	Não	
Nome	Texto	Não	Não	Não	

Fonte: Autoria própria.

No quadro 25 estão os campos da tabela Cidade.

Quadro 25- Campos da tabela Cidade

Campo	Tipo	Nulo	Chave primária	Chave estrangeira	Observações

IdCidade	Numérico	Não	Sim	Não	
Nome	Texto	Não	Não	Não	
UF	Texto	Não	Não	Não	

Fonte: Autoria própria.

No quadro 26, estão os campos da tabela Papel, que são os tipos de permissões dos usuários.

Quadro 26 - Campos da tabela Papel

Campo	Tipo	Nulo	Chave primária	Chave estrangeira	Observações
IdPapel	Numérico	Não	Sim	Não	
Nome	Texto	Não	Não	Não	

Fonte: Autoria própria.

No quadro 27 estão os campos da tabela Usuario_Papeis.

Quadro 27 - Campos da tabela Usuario_Papeis

Campo	Tipo	Nulo	Chave primária	Chave estrangeira	Observações
IdUsuario	Numérico	Não	Sim	Sim	Campo que vem da tabela Usuarios
IdPapel	Numérico	Não	Sim	Sim	Campo que vem da tabela Papel

Fonte: Autoria própria.

No quadro 28, estão os campos da tabela Residente_Glicemia.

Quadro 28 - Campos da tabela Residente_Glicemia

Campo	Tipo	Nulo	Chave primária	Chave estrangeira	Observações
IdResidenteGlicemia	Numérico	Não	Sim	Não	
IdResidente	Numérico	Não	Não	Sim	Campo que vem da tabela Residente
Valor	Numérico	Não	Não	Não	
QtdeInsulina	Numérico	Não	Não	Não	
DataAvaliacao	Date	Não	Não	Não	
HoraAvaliacao	Date	Não	Não	Não	

Observacao	Texto	Sim	Não	Não	
IdUsuario	Numérico	Não	Não	Sim	Da tabela Usuario
DataCriacao	Date	Não	Não	Não	

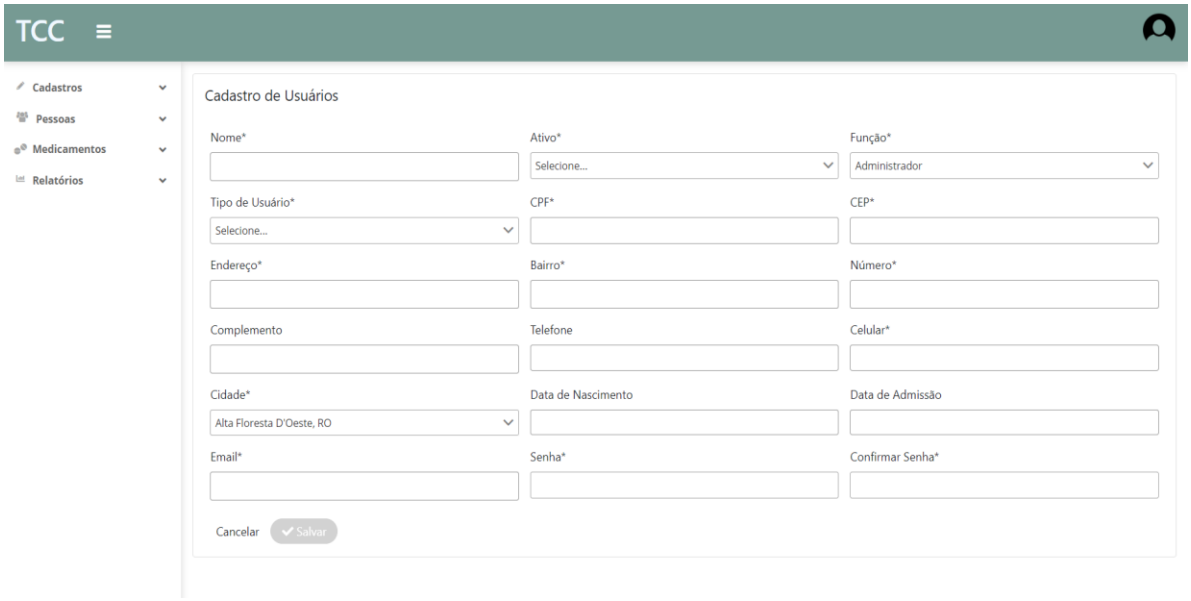
Fonte: Autoria própria.

4.3 APRESENTAÇÃO DO SISTEMA

De acordo com os requisitos levantados, foram criadas as telas, utilizando os componentes do Angular *PrimeNG*.

O sistema iniciará com uma conta de usuário administrador criada, que será repassada ao administrador do sistema. Por meio dessa conta, poderão ser criados todos os outros usuários do sistema. Os usuários dependerão que um administrador crie a sua conta. Na Figura 6 é apresentada a tela de cadastro de usuários.

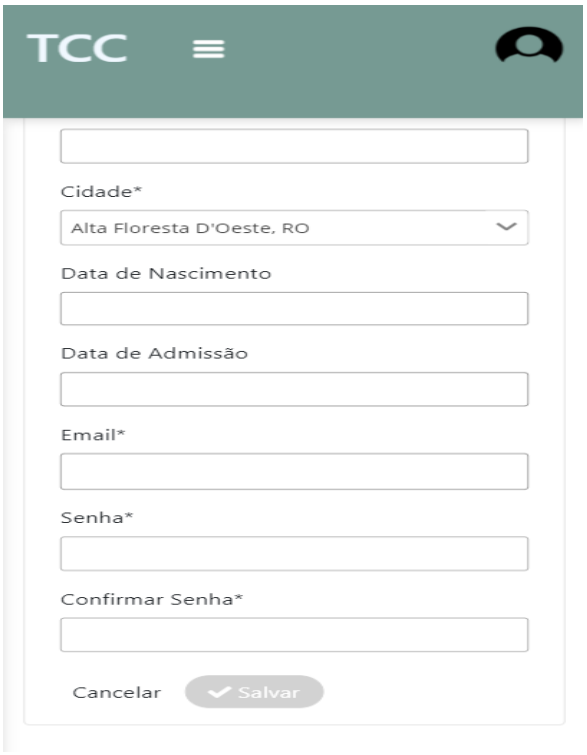
Figura 6 - Tela de cadastro de usuários



Fonte: Autoria própria.

Todas as telas da aplicação foram criadas visando a utilização em qualquer tipo de dispositivo, sendo abordado o conceito de responsividade. Na Figura 7 é apresentada a tela de cadastro de usuários para dispositivos móveis.

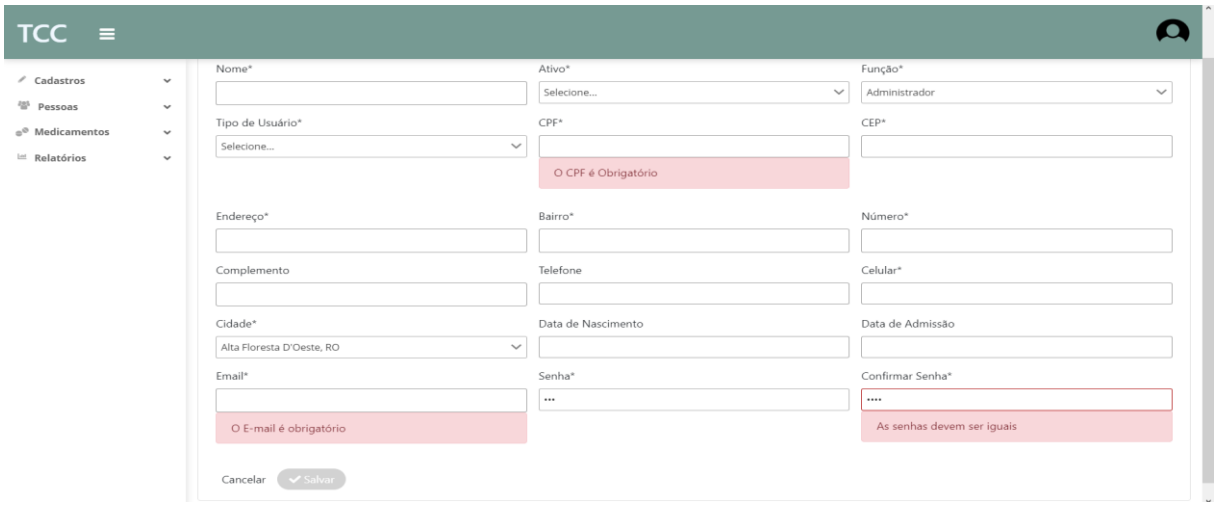
Figura 7 - Tela de cadastro de usuários em dispositivos móveis



Fonte: Autoria própria.

Conforme mostra a Figura 8, caso alguma informação obrigatória não seja informada ou algum campo seja preenchido de forma inválida, uma mensagem de *feedback* é mostrada ao usuário. E, também, enquanto todas as informações obrigatórias não estiverem preenchidas de forma correta, o botão "Salvar" não é habilitado.

Figura 8 - Validação dos campos no cadastro de usuários



Fonte: Autoria própria.

A Figura 9 apresenta a tela de autenticação do sistema, que é realizada utilizando o e-mail e a senha, previamente informados durante o cadastro de usuário.

Figura 9 - Tela de autenticação do sistema

Fonte: Autoria própria.

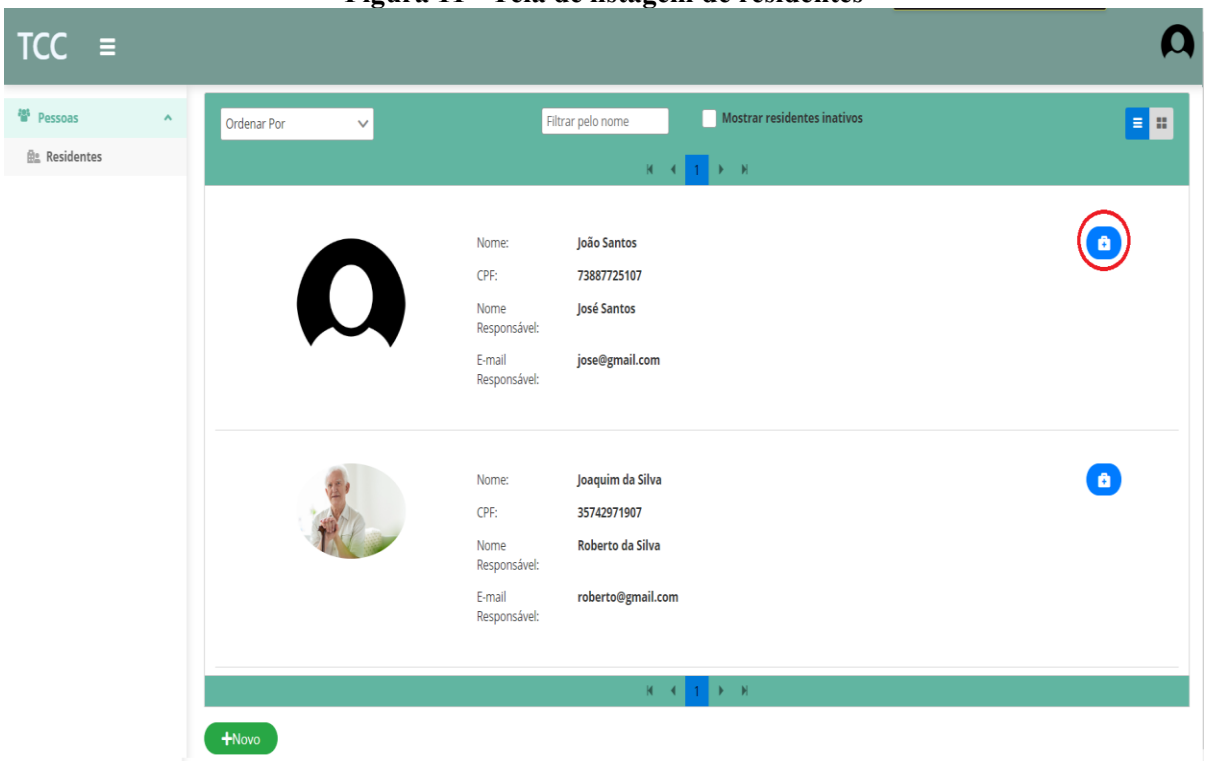
Na Figura 10 está a tela de autenticação para dispositivos móveis.

Figura 10 - Tela de autenticação em dispositivos móveis

Fonte: Autoria própria

Para inserir registros relacionados ao residente, o usuário deverá selecioná-lo na tela de listagem dos residentes, clicando no ícone clínico, que aparece circulado em vermelho na figura 11. Esse ícone será visível apenas para usuários do tipo profissional da saúde ou cuidador. Para usuários do tipo gestor e administrador, no lugar deste ícone, haverá um ícone para editar o cadastro do paciente ou deletá-lo A Figura 11 mostra a tela de listagem de residentes, logado com um usuário do tipo cuidador.

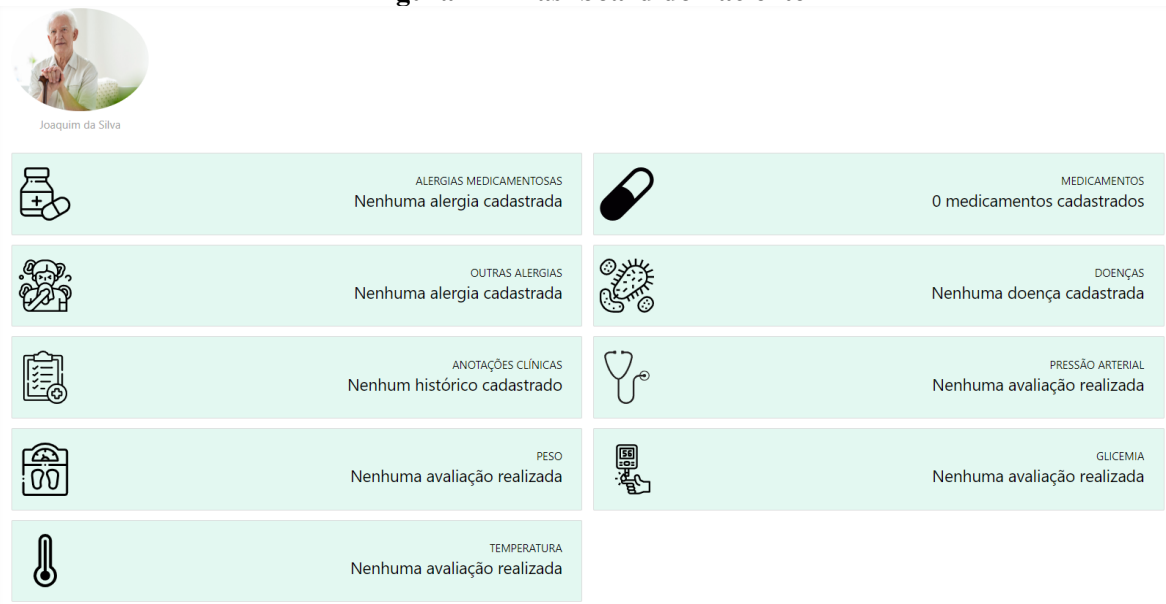
Figura 11 - Tela de listagem de residentes



Fonte: Autoria própria.

Ao clicar no ícone clínico do residente selecionado, será aberta uma página chamada de *dashboard*, que conterá um resumo de todos os dados relacionados a ele. Quando não houver informações cadastradas em determinado módulo, a *dashboard* exibirá que nenhuma informação foi inserida. Além disso, é por meio dessa tela que o usuário poderá cadastrar novas informações para o residente. A Figura 12 mostra a *dashboard* sem nenhuma informação cadastrada, em nenhum módulo.

Figura 12 - Dashboard do Paciente



Fonte: Autoria própria.

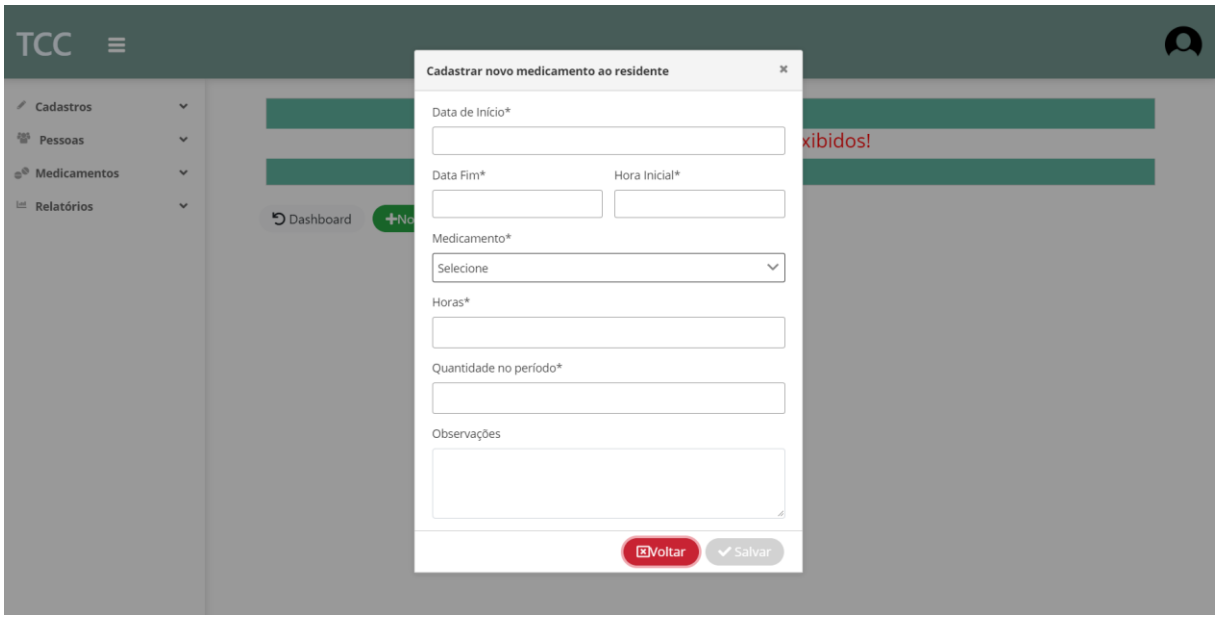
A Figura 13 mostra a *dashboard* em dispositivos móveis.

Figura 13 - Dashboard em dispositivos móveis

Fonte: Autoria própria.

Ao clicar em algum dos cartões da *dashboard*, como, por exemplo, em "Medicamentos", será aberta uma tela com a listagem de todos os medicamentos cadastrados para o residente. Nessa listagem, há um botão "Novo", ao clicar nesse botão, será aberto uma janela *modal* para incluir um novo medicamento para o residente. Enquanto não são preenchidos todos os campos obrigatórios no *modal*, o botão "Salvar" não é habilitado, conforme mostra a Figura 14.

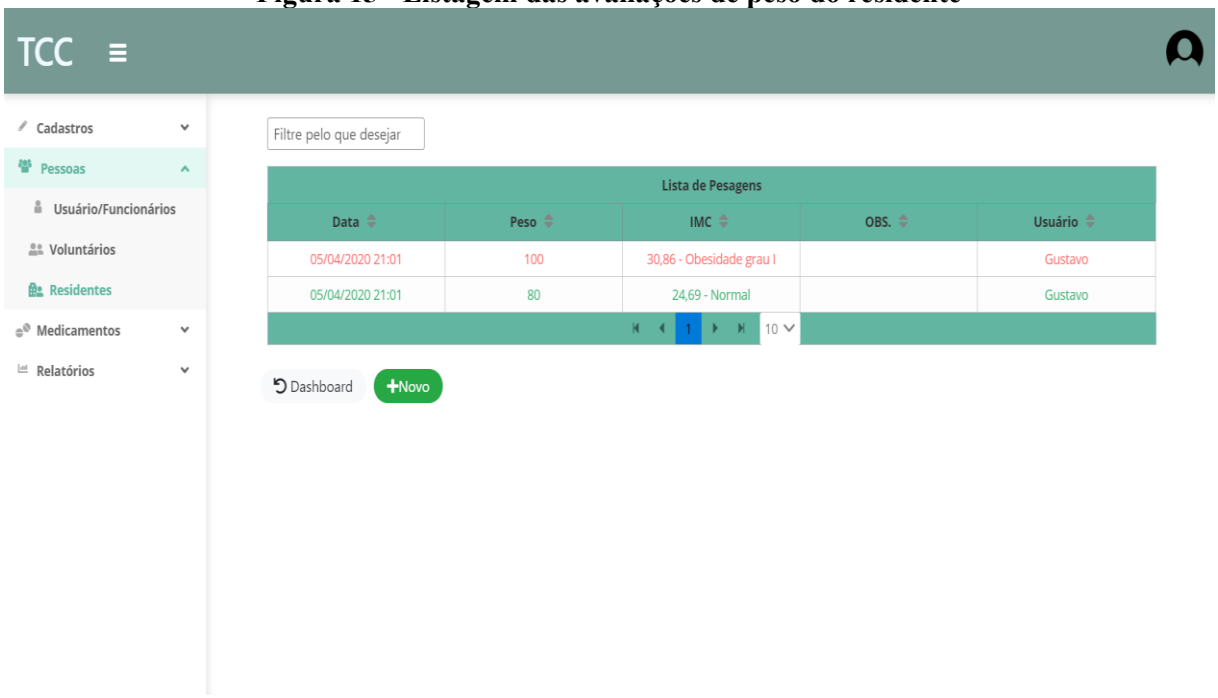
Figura 14 - Modal para inclusão de novo medicamento ao residente



Fonte: Autoria própria.

Em alguns casos, as listagens dos registros podem aparecer com a cor verde, vermelha ou amarela. Isso acontece nas avaliações de temperatura e peso. Com relação à temperatura, quando o residente estiver com febre alta, os registros serão exibidos em vermelho, quando o residente estiver apenas febril, os registros serão exibidos em amarelo e, quando a temperatura estiver normal os registros serão exibidos em verde. Com relação às avaliações de peso do residente, quando o IMC se apresentar acima ou abaixo do normal, os registros serão exibidos em vermelho, junto com o nível do IMC. Quando o IMC do residente estiver normal, os registros serão exibidos em verde. Isso está exemplificado na Figura 15. Para o cálculo do IMC, em pessoas menores de 60 anos foi utilizado uma classificação criada pela Organização Mundial da Saúde, já para pessoas com 60 anos ou mais, foi utilizada uma classificação criada por Lipschitz, que considera transformações ocorridas no processo de envelhecimento para realizar esse cálculo.

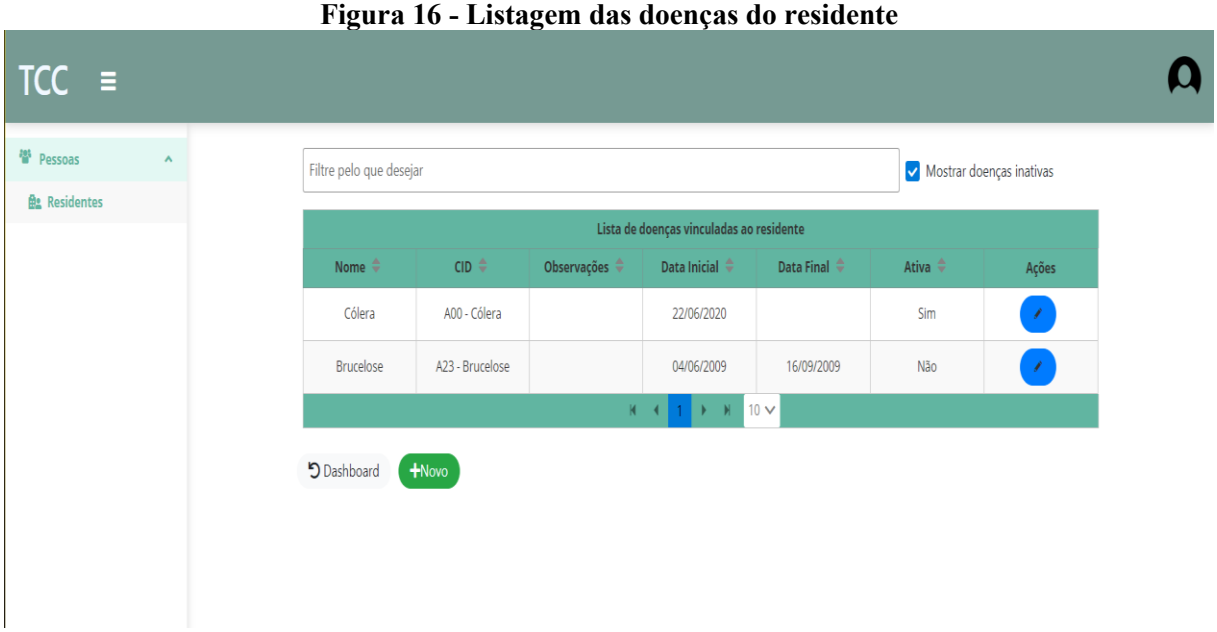
Figura 15 - Listagem das avaliações de peso do residente



Fonte: Autoria própria.

Em todas as listagens das informações dos residentes, há a opção de filtrar pelo campo que o usuário desejar e, também, ordenar da forma que ele achar necessário. Nos módulos em que há a opção de deixar o registro ativo ou inativo, como é o caso do cadastro de doenças, também há a opção de exibir ou não os registros inativos, como pode ser observado na Figura 16.

Figura 16 - Listagem das doenças do residente

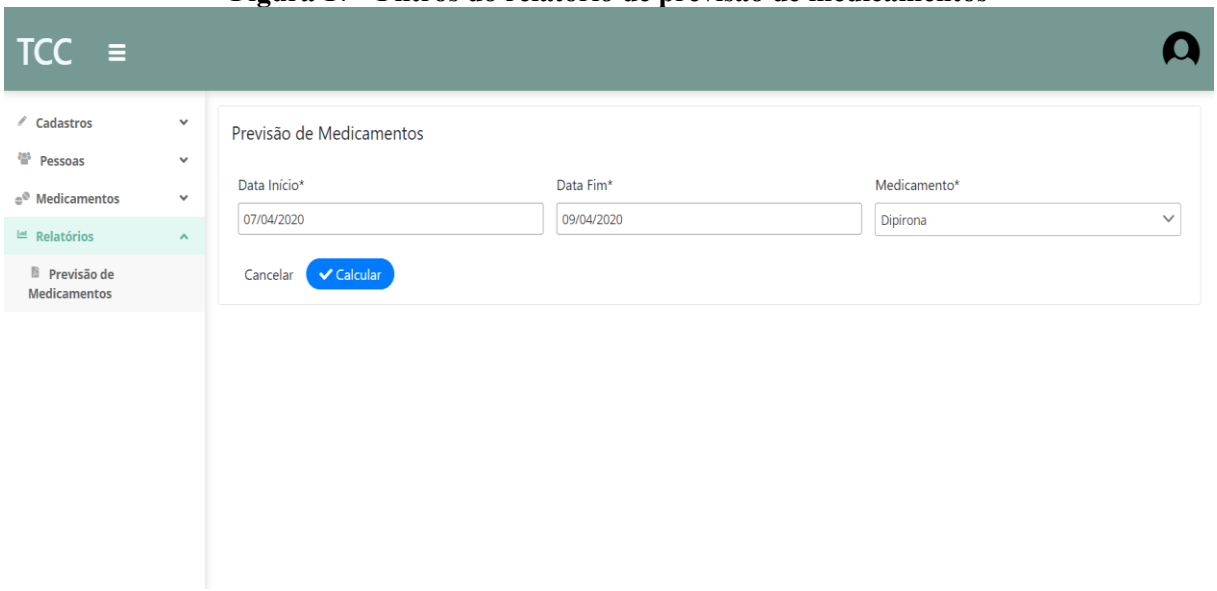


Fonte: Autoria própria.

O sistema também possui um relatório de previsões de medicamentos. Esse relatório fornece uma previsão da quantidade que será consumida de um determinado medicamento em um período. Essa quantidade é calculada baseando-se em todos os residentes que fazem ou farão uso desse medicamento durante o período filtrado e, também, na quantidade e a cada quanto tempo cada residente usará o medicamento. Por exemplo, ao cadastrar que o residente "1" utilizará um Dipirona a cada oito horas, do dia 07 até o dia 09, iniciando a partir da meia noite e o residente "2" fará uso de um dipirona a cada dezesseis horas, do dia 07 ao dia 09 também iniciando a meia noite, ao emitir o relatório, filtrando do dia 07 ao dia 09 o medicamento Dipirona, o relatório informará que durante o período filtrado, será necessária uma quantidade de doze desse medicamento, sendo que nove serão utilizados pelo residente 1 e três pelo residente 2. Esse filtro ajuda o gestor do asilo a saber a quantidade necessária para compra de um determinado medicamento em um determinado período.

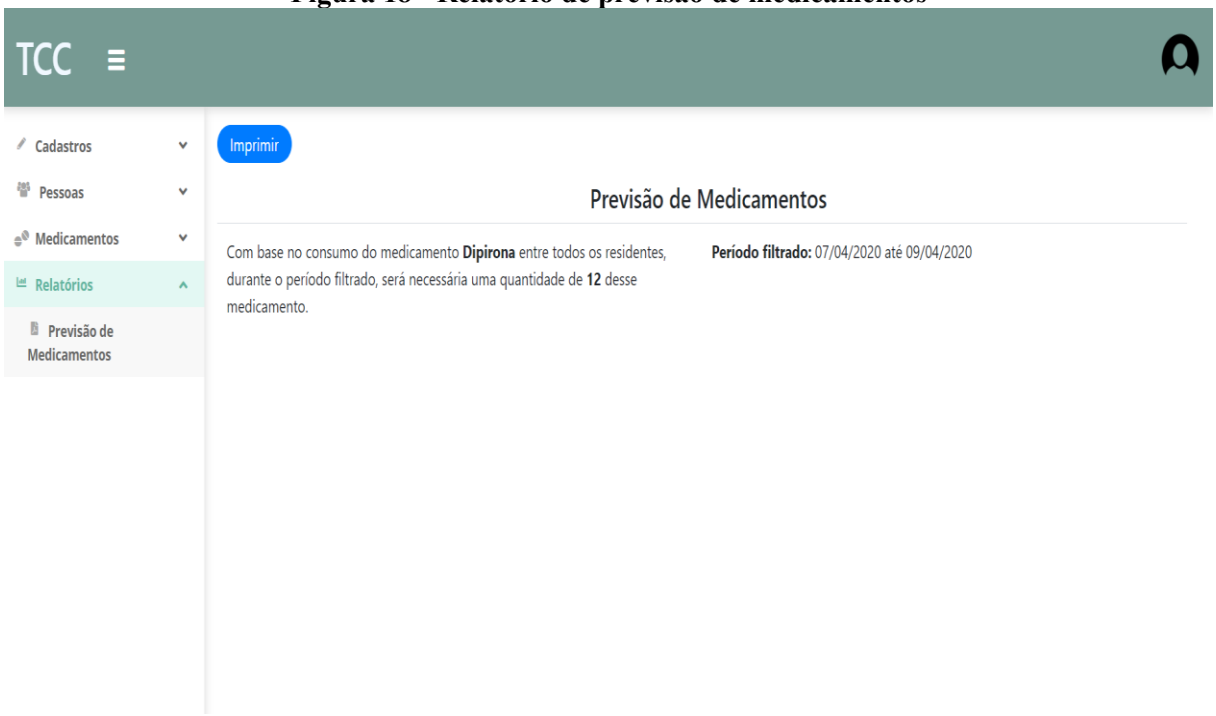
A Figura 17 mostra o filtro do relatório e a Figura 18 mostra o relatório.

Figura 17 - Filtros do relatório de previsão de medicamentos



Fonte: Autoria própria.

Figura 18 - Relatório de previsão de medicamentos



Fonte: Autoria própria.

4.4 IMPLEMENTAÇÃO DO SISTEMA

O projeto foi desenvolvido com uma estrutura que separa o lado servidor do lado cliente. Cada serviço é executado em uma porta diferente, podendo, também, serem executados em máquinas diferentes.

4.4.1 SERVIDOR

No lado servidor, foi utilizado o *framework* Spring. O Spring é um *framework* de desenvolvimento para Java. Para iniciar um novo projeto utilizando Spring, basta acessar o site https://start.spring.io, selecionar e preencher com as opções de preferência e gerar o projeto.

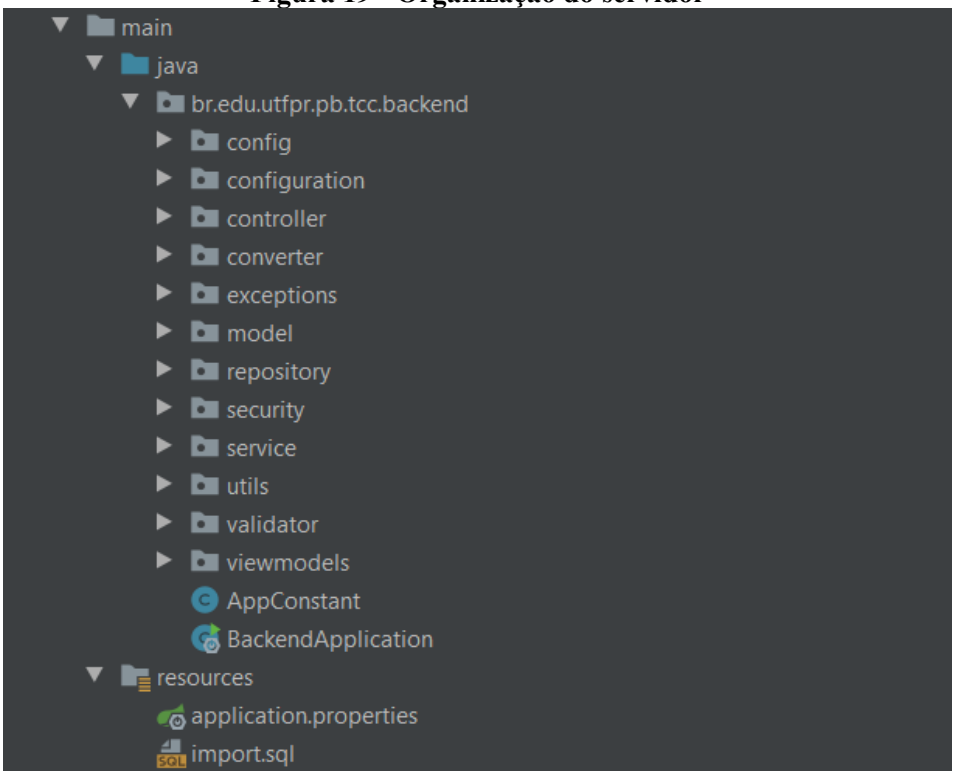
A organização das pastas do projeto é apresentada na Figura 19. No pacote *model* estão as entidades da aplicação. O *model* contém uma coleção de dados estruturados e, geralmente, correspondem a uma tabela no banco de dados. Eles podem ser acessados pelos *repositories, responses, controllers* e *services*. É nessa camada que ocorre a obtenção dos dados, convertendo-os em conceitos significativos para a aplicação, realizando também o processamento, validação e qualquer outra tarefa relativa ao tratamento dos dados.

Nos *repositories* estão objetos que isolam as entidades de domínio do código que acessa o banco de dados. Os *repositories* abstraem armazenamento e consulta de uma ou mais entidades de domínio.

Os *services* são objetos que possuem métodos que podem ser acessados em qualquer lugar da aplicação. Nos *services* são incluídas regras de negócio no sentido de tomar decisões e aplicar algoritmos de transformação de dados. Eles também são responsáveis por prover serviços diretamente a outras camadas ou módulos da aplicação.

Os *controllers* são responsáveis por responder requisições HTTP, geralmente agindo como intermediários entre os *models* e *views*. É a parte da aplicação que lida com as requisições do usuário. De certa forma, podem ser vistos como gerentes, cuidando para que todos os recursos necessários para completar uma determinada tarefa sejam delegados para os trabalhadores corretos. Ele aguarda os pedidos dos clientes, verifica a validade de acordo com as regras de autenticação e autorização, delega dados para serem obtidos ou processados pelos *models* e seleciona o tipo correto de apresentação dos dados para o qual o cliente está aceitando. Feito isso, delega o trabalho de renderização para a camada de visualização.

Figura 19 - Organização do servidor



Fonte: Autoria própria.

A primeira funcionalidade desenvolvida foi a autenticação de usuários. Essa autenticação foi desenvolvida utilizando JWT e o Spring Security. Na Listagem 1, está o código do *controller* que recebe as informações de autenticação inseridas pelo usuário no lado cliente. Esse controle designa o método responsável por buscar se as informações de autenticação inseridas pelo usuário estão corretas, validando o login e a senha. Essa busca é realizada utilizando o *model Usuario*. O *controller* chama o método *loadByUsername()* da interface *UsuarioService,* esse método instancia o método *findByUsername()* da classe *UsuarioRepository(),* que busca se o e-mail inserido pelo usuário existe na base.

Na etapa seguinte, é utilizada a biblioteca *brypt* para verificar se a senha do usuário está correta. Essa biblioteca também é utilizada para criptografar a senha do usuário na base ao editar ou incluir um novo usuário. Caso esteja tudo certo, é gerado um *token* de autenticação, por meio da classe *TokenUtils,* com algumas informações do login como e-mail, tipo da aplicação que irá usá-lo e data de criação. Posteriormente, esse *token* será utilizado para validar outras requisições realizadas pelo usuário. Este *token* é retornado pelo *controller* e a autenticação é realizada. Caso haja alguma informação incorreta com relação à autenticação, o *controller* retornará para o cliente o *status* 401, não autorizado.

Listagem 1 – *Controller* de autenticações

```
@RestController
@RequestMapping("auth")
public class AuthenticationController {
   @Autowired
   private AuthenticationManager authenticationManager;
   @Autowired
   private TokenUtils tokenUtils;

   @Autowired
   private UsuarioServiceImpl usuarioService;

   @PostMapping(value = "")
   public ResponseEntity<?> authenticationRequest(
         @RequestBody AuthenticationRequest a)
         throws AuthenticationException {

      Authentication authentication = this.authenticationManager.authenticate(
            new UsernamePasswordAuthenticationToken(a.getUsername(), a.getPassword())
      );

      SecurityContextHolder.getContext().setAuthentication(authentication);

      UserDetails userDetails = this.usuarioService.loadUserByUsername(a.getUsername());
      String token = this.tokenUtils.generateToken(userDetails);
      return ResponseEntity.ok(new AuthenticationResponse(token));
   }
```

```
    @GetMapping(value = "refresh")
    public ResponseEntity<?> authenticationRequest(
            HttpServletRequest request) {
        String token = request.getHeader(AppConstant.TOKEN_HEADER);
        String username = this.tokenUtils.getUsernameFromToken(token);

        Usuario u = (Usuario) this.usuarioService.loadUserByUsername(username);

        if (this.tokenUtils.canTokenBeRefreshed(token, u.getLastPasswordReset())) {
            String refreshedToken = this.tokenUtils.refreshToken(token);

            return ResponseEntity.ok(new AuthenticationResponse(refreshedToken));
        } else {
            return ResponseEntity.badRequest().body(null);
        }
    }

}
```

Fonte: Autoria própria.

Na Listagem 2 está o método *generateToken()* da classe TokenUtils, responsável por gerar o *token* do usuário quando a autenticação é realizada com sucesso. Este método recebe como parâmetro um *UserDetails*, que contém as informações do usuário autenticado. Neste método é criado um *HashMap* chamado *claims*. No primeiro par de elementos desse *HashMap* são passados como parâmetros a *string "sub"* e o login desse usuário, levando em conta que no *JWT "sub" (subject)* é a entidade a quem o *token* pertence. No segundo par de elementos são passados como parâmetros a *string "audience"* e a aplicação, no caso deste trabalho, uma aplicação *web*. No terceiro par de elementos são passados como parâmetros a *string "created"* e um método que retorna a data atual, representado a data que esse *token* foi gerado.

Listagem 2 - Método *generateToken()*

```
public String generateToken(UserDetails userDetails) {
    Map<String, Object> claims = new HashMap<>();
    claims.put("sub", userDetails.getUsername());
    claims.put("audience", this.AUDIENCE_WEB);
    claims.put("created", this.generateCurrentDate());

    return this.generateToken(claims);
}
```

Fonte: Autoria própria.

A Listagem 3 apresenta a classe *AuthenticationTokenFilter*. Para qualquer solicitação recebida, essa classe de filtro é executada. Ela verifica se a solicitação possui um *token JWT*

válido. Caso o *token* seja válido, ele define a autenticação no contexto para especificar que o usuário atual é autenticado.

Listagem 3 - Classe *AuthenticationTokenFilter*

```
public class AuthenticationTokenFilter extends UsernamePasswordAuthenticationFilter {


   @Autowired
   private TokenUtils tokenUtils;

   @Autowired
   private UsuarioServiceImpl userDetailsService;

   @Override
   public void doFilter(ServletRequest request,
                        ServletResponse response, FilterChain chain)
         throws IOException, ServletException {

      tokenUtils = WebApplicationContextUtils
            .getRequiredWebApplicationContext(this.getServletContext())
            .getBean(TokenUtils.class);
      userDetailsService = WebApplicationContextUtils
            .getRequiredWebApplicationContext(this.getServletContext())
            .getBean(UsuarioServiceImpl.class);

      HttpServletRequest httpRequest = (HttpServletRequest) request;
      HttpServletResponse resp = (HttpServletResponse) response;
      resp.setHeader("Access-Control-Allow-Credentials", "true");
      resp.setHeader("Access-Control-Allow-Origin", "*");
      resp.setHeader("Access-Control-Allow-Methods", "POST, PUT, GET, OPTIONS,
DELETE, PATCH");
      resp.setHeader("Access-Control-Allow-Headers", "Origin, X-Requested-With, " +
            "Authorization, Content-Type, Accept, X-CSRF-TOKEN, Cache-Control, DNT, " +
            "X-CustomHeader, Keep-Alive, User-Agent, If-Modified-Since, Content-Range, " +
            "Range, " + AppConstant.TOKEN_HEADER);
      resp.setHeader("Access-Control-Max-Age", "3600");

      if ("OPTIONS".equalsIgnoreCase(((HttpServletRequest) httpRequest).getMethod())) {
         resp.setStatus(HttpServletResponse.SC_OK);
      }

      String authToken = httpRequest.getHeader(AppConstant.TOKEN_HEADER);
      String username = this.tokenUtils.getUsernameFromToken(authToken);

      if (username != null && SecurityContextHolder.getContext().getAuthentication() == null) {

         UserDetails userDetails = this.userDetailsService.loadUserByUsername(username);

         if (this.tokenUtils.validateToken(authToken, userDetails)) {

            UsernamePasswordAuthenticationToken authentication =
                  new UsernamePasswordAuthenticationToken(userDetails,
                        null,
```

```
                        userDetails.getAuthorities());
                authentication.setDetails(
                        new WebAuthenticationDetailsSource().buildDetails(httpRequest));

                SecurityContextHolder.getContext().setAuthentication(authentication);
            }
        }
        chain.doFilter(request, response);
    }
}
```

Fonte: Autoria própria.

A parte de permissões é realizada pelo Spring Security, por meio da classe *WebSecurity*. As permissões são realizadas por meio de grupos de usuários. Neste trabalho, são divididos em quatro grupos, sendo eles: “admin”, “gestor”, “profsaude” e “cuidador”. Sempre que um usuário tentar acessar um determinado link, o Spring Security verificará se ele pertence a um grupo de usuário que possui permissão para isso. Conforme mostra a Listagem 4 que apresenta a classe *WebSecurity*, a URL desejada é passada por parâmetro no método *antMatchers().* Em seguida, é passado por parâmetro, no método *hasAnyRole(),* o nome do grupo de usuários que terá acesso a essa URL.

Listagem 4 - Classe WebSecurity

```
@Configuration
@EnableWebSecurity
@EnableGlobalMethodSecurity(prePostEnabled = true)
@EnableTransactionManagement
public class WebSecurityConfiguration extends WebSecurityConfigurerAdapter {

    @Autowired
    private UserDetailsService userDetailsService;
    @Autowired
    private AuthenticationEntryPoint authenticationEntryPoint;

    @Autowired
    public void configureAuthentication(
            AuthenticationManagerBuilder
                    authenticationManagerBuilder) throws Exception {
        authenticationManagerBuilder
                .userDetailsService(userDetailsService)
                .passwordEncoder(
                        new BCryptPasswordEncoder());
    }

    @Bean
    @Override
    public AuthenticationManager authenticationManagerBean()
            throws Exception {
        return super.authenticationManagerBean();
```

```
    }

    @Bean
    public AuthenticationTokenFilter
    authenticationTokenFilterBean() throws Exception {
        AuthenticationTokenFilter authenticationTokenFilter =
                new AuthenticationTokenFilter();
        authenticationTokenFilter.setAuthenticationManager(
                super.authenticationManagerBean()
        );
        return authenticationTokenFilter;
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.csrf().disable()
                .exceptionHandling()
                .authenticationEntryPoint(authenticationEntryPoint)
                .and()
                .sessionManagement()
                .sessionCreationPolicy(
                        SessionCreationPolicy.STATELESS)
                .and()
                .authorizeRequests()
                .antMatchers(HttpMethod.OPTIONS, "/**").permitAll()
                .antMatchers("/swagger-resources/**").permitAll()
                .antMatchers("/swagger-ui.html").permitAll()
                .antMatchers("/webjars/**").permitAll()
                .antMatchers("/v1/api-docs").permitAll()
                .antMatchers("/v2/api-docs").permitAll()
                .antMatchers("/session/**").authenticated()
                .antMatchers("/auth/**").permitAll()
                .antMatchers("/cidade").permitAll()
                .antMatchers("/cidade/busca").permitAll()
                .antMatchers("/cidade/**").hasAnyRole("ADMIN", "GESTOR")
                .antMatchers("/usuario/**").hasAnyRole("ADMIN", "GESTOR")
                .antMatchers("/papel/**").hasAnyRole("ADMIN")
                .antMatchers("/voluntario/**").hasAnyRole("ADMIN", "GESTOR")
                .antMatchers("/medicamento/buscaMedicamentos").permitAll()
                .antMatchers("/medicamento/**").hasAnyRole("ADMIN", "GESTOR")
                .antMatchers("/funcao/**").hasAnyRole("ADMIN", "GESTOR")
                .antMatchers("/principioAtivo/buscaPrincipiosAtivos").permitAll()
                .antMatchers("/principioAtivo/**").hasAnyRole("ADMIN", "GESTOR")
                .antMatchers("/pressaoArterial/**").hasAnyRole("PROFSAUDE", "CUIDADOR")
                .antMatchers("/residenteAlergiaMedic/**").hasAnyRole("PROFSAUDE",
"CUIDADOR")
                .antMatchers("/residenteAlergialGeral/**").hasAnyRole("PROFSAUDE",
"CUIDADOR")
                .antMatchers("/avaliacaoTemperatura/**").hasAnyRole("PROFSAUDE",
"CUIDADOR")
                .antMatchers("/historicoClinico/**").hasAnyRole("PROFSAUDE", "CUIDADOR")
                .antMatchers("/residenteDoenca/**").hasAnyRole("PROFSAUDE", "CUIDADOR")
                .antMatchers("/residenteMedicamento/**").hasAnyRole("PROFSAUDE",
"CUIDADOR")
                .anyRequest().permitAll();
```

```
        http.addFilterBefore(authenticationTokenFilterBean(),
                UsernamePasswordAuthenticationFilter.class);
    }
}
```

Fonte: Autoria própria.

Todos os *controllers* do sistema implementam a classe *CrudController,* que é uma classe que contém métodos genéricos para os *controllers* como, por exemplo, métodos para salvar, buscar todos os registros de um determinado módulo na base, buscar um registro em específico, deletar registros, entre outros. A Listagem 5 mostra a classe *CrudController*.

Listagem 5 - Classe *CrudController*

```
public abstract class CrudController <T, ID extends Serializable>{

  protected abstract CrudService<T, ID> getService();

  @GetMapping
  public List<T> findAll(){
    return getService().findAll();
  }

  @GetMapping("page")
  public Page<T> findAll(@RequestParam int page,
                         @RequestParam int size,
                         @RequestParam(required = false) String order,
                         @RequestParam(required = false) Boolean asc){
    PageRequest pageRequest = PageRequest.of(page, size);
    if (order != null && asc != null) {
      pageRequest = PageRequest.of(page, size, asc ? Sort.Direction.ASC : Sort.Direction.DESC,
order );
    }
    return getService().findAll(pageRequest);
  }

  @GetMapping("{id}")
  public T findOne(@PathVariable ID id) {
    return getService().findOne(id);
  }

  @PostMapping
  public T save(@RequestBody @Valid T entity) {
    return getService().save(entity);
  }

  @GetMapping("exists/{id}")
  public boolean exists(@PathVariable ID id) {
    return getService().exists(id);
  }
```

```
  @GetMapping("count")
  public long count() {
    return getService().count();
  }

  @DeleteMapping("{id}")
  public void delete(@PathVariable ID id) {
    getService().delete(id);
  }

}
```

Fonte: Autoria própria.

Quando for necessário utilizar um método diferente do método do *controller* genérico para salvar um registro, é criado um método no *controller* do módulo desejado, com uma rota diferente da rota que seria utilizada no método *save* genérico. Isso ocorre no método para salvar os usuários, em que precisam ser realizadas algumas validações antes que o objeto seja salvo como, por exemplo, verificar se o e-mail cadastrado existe na base. Essas validações são realizadas no método *saveWithValidation* na classe *UsuarioServiceImpl.* A Listagem 6 apresenta o método do *controller* para salvar usuários.

Listagem 6 - Método para salvar usuários

```
@PostMapping("novo")
@ResponseBody
public ResultadoOperacaoViewModel<Usuario> salvarUsuario(@RequestBody @Valid Usuario
source){
  return usuarioService.saveWithValidation(source);
}
```

Fonte: Autoria própria.

A Listagem 7 apresenta os métodos utilizados para cadastrar ou editar um usuário. Esses métodos estão na classe *UsuarioServiceImpl* e sobrescrevem os métodos da interface *UsuarioService*. O método *saveWithValidation* recebe a entidade *Usuario* do *UsuarioController.* Feito isso, esse método irá validar se o CPF ou o e-mail informado pelo usuário do sistema existem na base em algum outro cadastro. A validação do CPF, por exemplo, é realizada no método *findCpfCadastrado* que recebe o CPF por parâmetro e, por meio do método *existsByCpf* da classe *UsuarioRepository*, verifica se esse CPF está vinculado a outro cadastro. Caso esse CPF já exista, o cadastro desse usuário não será realizado e retornará uma mensagem de erro ao cliente.

Listagem 7 - Métodos para salvar e validar usuários no UsuarioServiceImpl

```
@Override
public ResultadoOperacaoViewModel<Usuario> saveWithValidation(Usuario entity) {
  var result = new ResultadoOperacaoViewModel<Usuario>();

  boolean idExist;
  if (entity.getId() != null){
    idExist = userRepository.existsById(entity.getId());
  } else{
    idExist = false;
  }

  var validate = new ValidaCpfCnpj();
  if(!validate.isCPF(entity.getCpf())) {
    return result.returningFailMessage(ValidationMessages.FormatoCpfInvalido);
  }

  var checkCadastrado = this.findCpfCadastrado(entity.getCpf());
  if(checkCadastrado.isSucesso() && !idExist){
    return result.returningFailMessage(ValidationMessages.PessoaJaExiste);
  }

  var checkUsuario = this.findUsernameCadastrado(entity.getUsername());
  if(checkUsuario.isSucesso() && !idExist){
    return result.returningFailMessage(ValidationMessages.UsuarioJaExiste);
  }


  if(entity.getPassword().length() <= 18){
    entity.setPassword(entity.getEncodedPassword(entity.getPassword()));
  } else{
    entity.setPassword(entity.getPassword());
  }

  super.save(entity);

  return result.returningSuccess(entity);
}

@Override
public ResultadoOperacaoViewModel<Object> findCpfCadastrado(String cpf) {
  var result = new ResultadoOperacaoViewModel<>();

  var exists = userRepository.existsByCpf(cpf);

  if (!exists) {
    return result.returningFailMessage(ValidationMessages.CpfNaoExiste);
  }

  return result.returningSuccess();
}

@Override
public ResultadoOperacaoViewModel<Object> findUsernameCadastrado(String username) {
  var result = new ResultadoOperacaoViewModel<>();
```

```
  var exists = userRepository.existsByUsername(username);

  if (!exists) {
    return result.returningFailMessage(ValidationMessages.UsuarioNaoExiste);
  }

  return result.returningSuccess();
}
```

Fonte: Autoria própria.

Grande parte das APIs criadas no sistema utilizam o padrão implementado no Spring, porém existem várias regras de negócio aplicadas que foram desenvolvidas para atender necessidades específicas do sistema, como as validações do usuário que foram citadas nas listagens 6 e 7, por exemplo.

4.4.2 CLIENTE

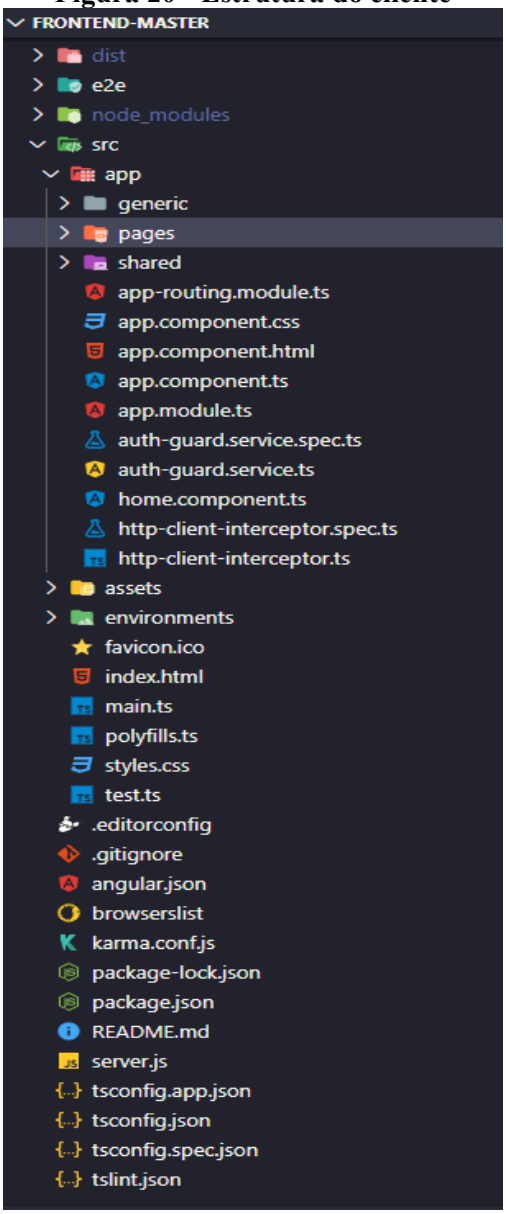
Do lado cliente foi utilizado o *framework* Angular. Em suas versões mais recentes, o mesmo oferece apoio da ferramenta AngularCli, que foidesenvolvida pela própria equipe do Angular e é utilizada para facilitar a criação de componentes, classes, *services*, entre outros.

Para gerenciamento de pacotes, foi utilizado o Node Package Manager (NPM). Basicamente, o NPM é uma ferramenta de linha de comando que ajuda a interagir com plataformas *on-line*, como servidores. Ele auxilia na instalação e desinstalação de pacotes e gerenciamento de dependências necessárias para executar um projeto. Para usá-lo, é necessário apenas instalar o node.js.

Para instalar o AngularCli, é necessário o Node 6.9.0 ou superior e o NPM 3 ou superior. A instalação é simples, bastando executar o comando “npm install -g @angular/cli” no terminal. Feito isso, para criar um projeto inicial em Angular utilizando o AngularCli, é necessário executar o comando “ng new nomedoprojeto”.

O projeto cliente deste trabalho possui a estrutura apresentada na Figura 20.

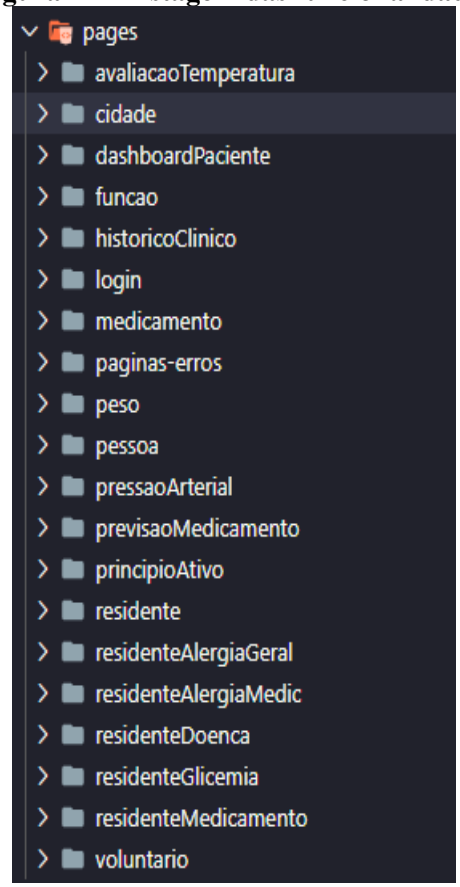
Figura 20 - Estrutura do cliente



Fonte: Autoria própria.

Cada funcionalidade do projeto foi separada em subpastas dentro da pasta *pages*, conforme mostra a Figura 21.

Figura 21 - Listagem das funcionalidades



Fonte: Autoria própria.

Para o desenvolvimento das interfaces foi utilizado o *framework* PrimeNG. A utilização desse *framework* teve um papel muito importante no desenvolvimento do *frontend*, pois trouxe um ganho muito grande de tempo e trabalho. A maioria dos componentes nas telas da aplicação são oriundas desse *framework*.

A declaração dos *components* internos da aplicação, *providers* de *services* e a importação dos modules externos são feitas no módulo principal. Esse módulo é apresentado de forma resumida na Listagem 8.

Listagem 8 - Listagem do módulo principal

```
import { NgModule } from "@angular/core";
...
import { MessageService } from "primeng/api";
...
import { AppRoutingModule } from "./app-routing.module";
import { AppComponent } from "./app.component";
...
import { CidadeModule } from "./pages/cidade/cidade.module";
...
import { HomeComponent } from "./home.component";
...
```

```
@NgModule({
   declarations: [AppComponent, LoginComponent, HomeComponent],
   imports: [
      BrowserModule,
      CardModule,
      ToastModule,
      CheckboxModule,
      ...
      PesoModule,
      AvaliacaoTemperaturaModule,
      PressaoArterialModule,
      ResidenteAlergiaGeralModule,
      ...
      AppReportModule,
      PrincioAtivoModule,
      ...
      ButtonModule,
      MensagemModule,
      UsuarioModule,
   ],
   providers: [
      MessageService,
      LoginService,
      AppComponent,
      {
         provide: HTTP_INTERCEPTORS,
         useClass: HttpClientInterceptor,
         multi: true
      },
      AuthGuardService
   ],
   exports: [HomeComponent],
   bootstrap: [AppComponent]
})
export class AppModule {}
```

Fonte: Autoria própria.

As questões de roteamento do cliente são definidas a partir do arquivo app.routing.ts. O *array* de rotas apresentado na Listagem 9 define como ocorre a navegação entre as URLs do sistema. Cada rota define um *component* para uma determinada URL.

Na Listagem 9, existe um pai com o atributo *path* vazio e outro com o *path* “login”. O que possui o *path* vazio, também possui diversas rotas filhas. Todas essas rotas filhas dependem que o usuário esteja autenticado para que o mesmo possa acessá-las. Levando em conta que todas as funcionalidades dependem de um usuário autenticado, obviamente, a única rota que não dependerá disso é a de login.

Para uma das rotas de medicamento, por exemplo, há a opção de parâmetros, onde o *id* de um determinado registro passa a ser um parâmetro de rota. Isso pode ser utilizado para

editar ou visualizar um medicamento com o id selecionado pelo usuário onde, se selecionado o medicamento cujo id é 10, a URL ficaria /medicamento/10.

A rota com path "**" serve para que o roteador selecione o componente indicado nessa rota sempre que alguma URL digitada pelo usuário não exista na aplicação.

Listagem 9 - Arquivo *app-routing.module.ts*

```
const routes: Routes = [
  {
    path: "",
    canActivate: [LoginService],
    children: [
      {
        path: "",
        component: HomeComponent,
        redirectTo: "",
        pathMatch: "full"
      },
      { path: "cidade", component: CidadeListComponent },
      { path: "cidade/novo", component: CidadeFormComponent },
      { path: "cidade/:id", component: CidadeFormComponent },
      { path: "usuario", component: UsuarioListComponent },
      { path: "usuario/novo", component: UsuarioFormComponent },
      { path: "usuario/:id", component: UsuarioFormComponent },
...
      { path: "medicamento", component: MedicamentoListComponent },
      { path: "medicamento/novo", component: MedicamentoFormComponent },
      { path: "medicamento/:id", component: MedicamentoFormComponent },
...
      { path: "residente/:id/peso", component: PesoListComponent },
      { path: "residente/:id/peso/novo", component: ResidenteMedicamentoFormComponent },
      { path: "residente/:id/peso/:id/novo", component: ResidenteMedicamentoFormComponent },
...
      { path: "residente/:id/residenteDoenca", component: ResidenteDoencaComponent },
      { path: "residente/:id/historicoClinico", component: HistoricoClinicoComponent },
      { path: "residente/:id/residenteGlicemia", component: ResidenteGlicemiaComponent },
      { path: "previsaoMedicamento", component: PrevisaoMedicamentoComponent },
      { path: "reportPrevisaoMedicamento", component: AppReportComponent },
    ]
  },
  { path: "login", component: LoginComponent },
  { path: "**", component: Pagina403Component },
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
// @ts-ignore
export class AppRoutingModule { }
```

Fonte: Autoria própria.

A página de autenticação é sempre exibida ao tentar acessar alguma URL da aplicação sem estar logado. A construção da mesma foi realizada por meio dos códigos presentes nas listagens 10 e 11. Sempre que o usuário clicar em “Logar”, os dados do formulário de login serão enviados ao *component* e chamará o método login(), que também pertence ao *component*. Sempre que esse método for chamado, ele invocará o método login() da classe LoginService, passando como parâmetros o e-mail e a senha informados pelo usuário. Esse método será responsável pela comunicação do cliente com o servidor. O mesmo chamará o AuthController, do lado do servidor, por meio da URL “/auth”, repassando o e-mail e a senha como um JSON. Caso o controller retorne que está tudo certo com as informações repassadas pelo usuário, o mesmo será autenticado, seu *token* será adicionado ao LocalStorage e será redirecionado a tela inicial da aplicação. Caso alguma informação inserida por ele esteja incorreta, o mesmo não será autenticado e o cliente exibirá uma mensagem de erro na tela de login. A classe LoginService é apresentada na Listagem 12.

Listagem 10 - Código *HTML* da página de login

```
<div class="" id="container">
        <div class="form-container sign-in-container">
                <form [formGroup]="form">
                        <div class="container">
                                <div id="titleLogin">
                                        <h1>Seja Bem-Vindo</h1>
                                        <span>Preencha corretamente o formulário para acessar sua
conta</span>
                                </div>
                                <div>
                                        <img id="img-login" src="assets/images/img-login.png">
                                </div>

                                <div id="teste">
                                        <input id="username" formControlName="username"
type="email" placeholder="Email" />
                                </div>
                                <input id="password" formControlName="password"
type="password" placeholder="Senha" />
                                <button (click)="login()">Logar</button>
                        </div>
                </form>
        </div>
</div>
<p-toast></p-toast>
```

Fonte: Autoria própria.

Listagem 11 - Classe *components* do login

```
@Component({
  selector: 'app-login',
  templateUrl: './login.component.html',
  styleUrls: ['./login.component.css'],
  providers: [MessageService]
})
export class LoginComponent implements OnInit {

  msgs: Message[] = [];
  form: FormGroup;

  @Input() usuarioFoiAutenticado: boolean = false;
  @Output() mostrarMenu: EventEmitter<any> = new EventEmitter();
  @Output() ocultarMenu: EventEmitter<any> = new EventEmitter();

  constructor(private loginService: LoginService,
    private router: Router,
    private messageService: MessageService) {
    this.form = new FormBuilder().group({
      username: ['', Validators.required],
      password: ['', Validators.required],
    });
  }

  ngOnInit() {
    const token = localStorage.getItem('token');

    if (token) {
      this.router.navigate(['']);
    }

  }

  login() {
    this.loginService.login(this.form.controls.username.value,
this.form.controls.password.value).subscribe(e => {
      this.router.navigate(['/']);
    }, error => {
      this.messageService.add({ severity: 'error', summary: 'Error', detail: 'Usuário e/ou senha
incorreto(s)!' });

      this.msgs = [{
        severity: 'error', summary: 'Error',
        detail: 'Usuário e/ou senha incorreto(s)!'
      }];
    });
  }
}
```

Fonte: Autoria própria.

Listagem 12 - Classe *LoginService*

```
@Injectable({
  providedIn: "root"
})
export class LoginService {
  userInfo: any;
  isAuthenticated = new Subject<boolean>();
  isRunningRequest: boolean = false;

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

  canActivate(
    route: ActivatedRouteSnapshot,
    state: RouterStateSnapshot
  ): Observable<boolean> | Promise<boolean> | boolean {
    const url = `${environment.api}/session/user-info`;
    this.isRunningRequest = true;

    return this.http.get(url).pipe(
      map(e => {
        this.userInfo = e;
        this.isRunningRequest = false;
        this.isAuthenticated.next(true);
        return true;
      }),
      catchError(err => {
        this.isRunningRequest = false;
        this.logout();
        return throwError(new Error("O usuário não está autenticado!"));
      })
    );
  }

  async getUserInfo(): Promise<any> {
    return new Promise<any>(async res => {
      while (this.isRunningRequest) {
        await new Promise(resolve => {
          setTimeout(() => {
            resolve();
          }, 100);
        });
      }
      res(this.userInfo);
    });
  }
...
  logout() {
    localStorage.removeItem("token");
    this.isAuthenticated.next(false);
    this.router.navigate(["/login"]);
  }

  login(
    username: string,
```

```
      password: string
    ): Observable<AuthenticationResponse> {
      const request = new AuthenticationRequest();
      request.username = username;
      request.password = password;
      const headers = new HttpHeaders({
        "Content-type": "application/json",
        Accept: "application/json"
      });
      return this.http
        .post<AuthenticationResponse>(
          `${environment.api}/auth/`,
          JSON.stringify(request),
          { headers }
        )
        .pipe(
          map(e => {
            Object.keys(e).forEach(key =>
              localStorage.setItem(key, e[key])
            );
            this.isAuthenticated.next(true);
            return e;
          })
        );
    }
...
}
```

Fonte: Autoria própria.

Na classe *LoginService* também há outros métodos importantes para a aplicação como, por exemplo, o *getUserInfo().* Esse método é utilizado no *component* do menu da aplicação onde, através dele é obtido o grupo de usuários do usuário autenticado para validar a quais menus ele pode ter acesso.

A Listagem 13 apresenta, de forma resumida, o *component* do menu. Nesta classe, é criado um *array* chamado rotas, é nesse *array* que definimos os menus do sistema e suas características. Dentro de cada menu há o *label,* que define o nome do menu, o item *roles* define qual grupo de usuário terá esse menu disponível, o item *icon,* que define o ícone que este menu terá. Todos os menus possuem submenus, que são definidos dentro do *items* de cada menu. A URL que cada submenu redireciona é definida em *routerlink.*

A validação sobre quais menus o usuário logado terá acesso acontece no método *getUserInfo().* Nesse método, a aplicação verifica se o grupo do usuárioautenticado pertence a um dos grupos definidos no item *roles* de cada um dos menus ou submenus. Caso o mesmo pertença a um desses grupos, esse menu é adicionado em um *array* chamado itensMenu. É

esse *array* que a aplicação utiliza para montar a lista de menus que serão exibidos, conforme apresenta a Listagem 14.

Listagem 13 - *Component* do menu

```
@Component({
   selector: "app-menu",
   templateUrl: "./menu.component.html",
   styleUrls: ["./menu.component.css"]
})
export class MenuComponent implements OnInit, AfterViewInit {
   @Input() reset: boolean;
   itensMenu: MenuItem[];
   rotas: any[] = [
...
          {
          label: "Pessoas",
          roles: ["ADMIN"],
          icon: "pi pi-fw pi-users",
          items: [
             {
                label: "Usuário/Funcionários",
                icon: "pi pi-fw pi-user",
                routerLink: "usuario",
                roles: ["ADMIN"]
             },
             {
                label: "Voluntários",
                icon: "fas fa-user-friends",
                roles: ["ADMIN"],
                routerLink: "voluntario"
             },
             {
                label: "Residentes",
                icon: "fas fa-hospital-user",
                roles: ["ADMIN"],
                routerLink: "residente"
             },
```

```
        ]
      },
...
  ];
  @ViewChild('layoutMenuScroller', { static: true }) layoutMenuScrollerViewChild: ScrollPanel;
  constructor(private loginService: LoginService) {}
  ngOnInit() {

    this.itensMenu = [];

    this.loginService.getUserInfo().then(info => {
      const userRoles = info.authorities.map((x: any) =>
        x.nome.replace("ROLE_", "")
      );
      this.rotas.forEach((r: any) => {
        let routeAux = null;

        if (
          r.roles.filter(value => -1 !== userRoles.indexOf(value))
            .length > 0
        ) {
          routeAux = r;
          let items = [];

          if (routeAux.items) {
            routeAux.items.forEach((childRoute: any) => {
              if (
                childRoute.roles.filter(
                  value => -1 !== userRoles.indexOf(value)
                ).length > 0
              ) {
                items.push(childRoute);
              }
            });
          }
          routeAux.items = items;
          this.itensMenu.push(routeAux);
```

```
            }
        });
    });
  }
}
...
```

Fonte: Autoria própria.

Listagem 14 - Classe em que é montado o HTML dos menus

```
<div class="layout-menu-container">
  <p-scrollPanel #layoutMenuScroller>
      <div class="menu-scroll-content">
          <ul app-submenu [item]="itensMenu" root="true" class="layout-menu"
visible="true" [reset]="reset" parentActive="true"></ul>
      </div>
  </p-scrollPanel>
</div>
```

Fonte: Autoria própria.

O padrão para as telas do sistema é a realização das ações de CRUD que são: salvar, editar, excluir e listar dados. Para facilitar isso, foi criado um *service* genérico, para realizar operações básicas de um CRUD. Esse *service* possui uma constante chamada *httpOptions*, que é utilizada para criar o *header* das requisições HTTP que serão encaminhadas ao servidor.

Cada um dos métodos que esse *service* possui retornam um *Observable* com o tipo desejado. O método *delete(),* por exemplo, realiza uma requisição com método HTTP do tipo *DELETE* para a URL desejada, junto com o *id* que deseja-se excluir.

Para salvar um novo registro, o método *save* espera uma entidade como parâmetro e tem como retorno um *Observable* do tipo *POST.* É necessário enviar uma requisição HTTP com o método *POST,* para a URL desejada, passando o objeto como um *JSON.* Caso tenha sucesso, o servidor retorna o registro salvo e, assim, o *service* retorna para o *component* que realizou a chamada. A Listagem 15 apresenta, de forma resumida, o *service* genérico criado para a aplicação.

Listagem 15 - *Service* de CRUD genérico

```
...
export const httpOptions = {
    headers: new HttpHeaders({
        'Content-Type': 'application/json',
        Accept: 'application/json'
    })
};
```

```
export abstract class CrudService<T, ID> {

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

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

    findAll(): Observable<T[]> {
        const url = `${this.getUrl()}`;
        return this.http.get<T[]>(url);
    }
...
    save(t: T): Observable<T> {
        const url = `${this.getUrl()}`;
        return this.http.post<T>(url, JSON.stringify(t), httpOptions);


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

Fonte: Autoria própria.

Um exemplo de como funcionam os CRUDs da aplicação é o *component* de cadastro de doenças, que pode ser observado, de forma resumida, na Listagem 16. O *ResidenteDoencaComponent* implementa, entre outras coisas, a classe *OnInit* que possui o método *ngOnInit(),* que é executado após os componentes da tela serem inicializados. No caso deste *component,* estão sendo carregadas todas as doenças que pertencem ao residente selecionado e, também, todos os CIDs cadastrados na base. Esses métodos se comportam da mesma maneira, cada um chama um método de seu respectivo *service,* que irá realizar as requisições no servidor.

Listagem 16 - *ResidenteDoencaComponent*

```
@Component({
   selector: "app-residenteDoenca-list",
   templateUrl: "./residenteDoenca.component.html",
   styleUrls: ["./residenteDoenca.component.css"]
})

...

export class ResidenteDoencaComponent implements OnInit {

residenteDoenca: ResidenteDoenca[];

cidsOption: LabelValue[] = [];
```

```
cols: any[];

display: boolean;
form: FormGroup;

 constructor(

      private residenteDoencaService: ResidenteDoencaService,
      protected router: Router,
      private route: ActivatedRoute,
      private toastService: ToastService,
      private cidService: CidService
   )

...

  ngOnInit() {
      this.findByResidenteID();
...
      this.cols = [
          { field: "nome", header: "Alergia" },
          { field: "cid.codigo", header: "CID" },
          { field: "observacao", header: "Observações" },
          { field: "ativo", header: "Ativa" }
      ];
      this.buscaCids();
      this.valueCheckBoxAtivo = false;
   }

...

   findByResidenteID() {
      this.residenteDoencaService
          .findByResidenteID(this.route.snapshot.params.id)
          .subscribe(
              residenteDoencas => {
                  this.residenteDoenca = residenteDoencas;
              },
              err => {
                  if (err["status"] == 403) {
                      this.router.navigate(["pagina-403"]);
                  }
                  if (err["status"] == 404) {
                      this.router.navigate(["pagina-403"]);
                  }});
}
buscaCids() {
      this.cidService.findAll().subscribe(
          cids => {
              this.cidsOption =
                  cids.map(cid => ({
                      label: `${cid.codigo} - ${cid.nome}`,
                      value: cid
                  })) || [];
          },
          error => {
              this.toastService.showError(
                  "Atenção",
                  "Não foi possível buscar os Cids"
```

```
            );
    });
    }
}
```

Fonte: Autoria própria.

Para que seja possível abrir e fechar o *dialog,* foi criada uma variável *boolean* chamada *display* e um evento de *click.* Sempre que o usuário executar alguma ação para abrir ou fechar o *dialog*, o evento de *click* será chamado e alterará a variável *display* para verdadeiro quando for para abrir o mesmo ou falso quando for para fechar.

Listagem 17 - Evento de click ao clicar no botão Salvar

```
<button type="button" class="btn btn-success" [disabled]="!form.valid"
(click)="save()" (click)="display=false"> <i class="fa fa-check"> </i> Salvar
</button>
```

Fonte: Autoria própria.

O *ResidentedoencaComponent* também possui um *array* chamado *cols.* É nesse *array* que definimos quais itens serão apresentados na *table,* onde o item *field* corresponde ao campo do banco que buscaremos as informações e o item *header* corresponderá ao nome da *th.* A Listagem 18 mostra como a *table* é exibida ao usuário, utilizando o objeto *ResidenteDoenca.* A quantidade de itens exibidos por página é definida por meio do *rowsPerPageOptions,* onde poderão ser exibidos cinco, dez, quinze ou vinte registros por página. Nessa *table* os registros podem ser ordenados de diferentes formas, sempre que o usuário clicar no *sortIcon.*

Listagem 18 - *Table* que exibe as doenças dos residentes

```
...
<p-table #dv [columns]="cols" [value]="residenteDoenca" selectionMode="single" filterBy="nome"
      [paginator]="true" [rows]="10" [responsive]="true" [rowsPerPageOptions]="[5,10,15,20]">

  <ng-template pTemplate="caption">
    Lista de doenças vinculadas ao residente
  </ng-template>
  <ng-template pTemplate="header" let-columns>
    <tr align="center" *ngIf="!verificaVazio()">
      <th *ngFor="let col of columns" [pSortableColumn]="col.field">
        {{col.header}}
        <p-sortIcon [field]="col.field"></p-sortIcon>
      </th>
      <th>Ações</th>
    </tr>
    <div align="center" *ngIf="verificaVazio()" style="color: red;">
```

```
        <h4 align="center">Não há dados para serem exibidos!</h4>
      </div>
    </ng-template>
    <ng-template pTemplate="body" let-residenteDoenca let-columns="columns">
      <tr align="center" *ngIf="!verificaVazio()">
        <td>
          <span class="ui-column-title">Alergia</span>
          {{residenteDoenca.nome}}
        </td>
        <td>
          <span class="ui-column-title">CID</span>
          {{residenteDoenca.cid.codigo}} - {{residenteDoenca.cid.nome}}
        </td>
        <td>
          <span class="ui-column-title">Observações</span>
          {{residenteDoenca.observacao}}
        </td>
        <td>
          <span class="ui-column-title">Ativa</span>
          {{residenteDoenca.ativo ? "Sim" : "Não"}}
        </td>
        <td>
          <span class="ui-column-title">Ações</span>
          <a (click)="editClick(residenteDoenca.id)" (click)="display=true" class="btn btn-primary
mr-2 pi pi-pencil"></a>
        </td>
      </tr>
    </ng-template>
</p-table>

...
```

Fonte: Autoria própria.

Praticamente todas as demais telas do sistema, principalmente as que se referem aos dados de saúde dos residentes, seguem um fluxo parecido ao apresentado no *ResidenteDoencaComponent.*

5 CONCLUSÃO

Neste trabalho foi desenvolvida uma solução computacional para asilos de idosos, com o principal objetivo de agilizar e facilitar as atividades realizadas pelas pessoas que trabalham nestes locais, ajudando no controle dos dados clínicos dos residentes. Para isso, foi desenvolvida uma aplicação utilizando os conceitos de REST API.

No *back-end* foi utilizada a linguagem Java para desenvolvimento, com o auxílio do *framework* Spring. Sem dúvidas a utilização deste *framework* auxiliou muito no desenvolvimento do projeto, permitindo que o *back-end* da aplicação fosse desenvolvido de maneira mais ágil e consumindo menos tempo.

Na parte do *front-end,* foram utilizados os *frameworks* Angular e PrimeNG. A utilização principalmente do PrimeNG, também possibilitou um enorme ganho de tempo no trabalho, visto que todas as telas do sistema utilizam algum dos seus componentes.

As maiores dificuldades encontradas no desenvolvimento da solução foram com relação ao desenvolvimento do *front-end*, visto que, quando o trabalho foi iniciado, não possuía tanto conhecimento em Angular e em TypeScript. Assim, o desenvolvimento deste sistema resultou em um grande aprendizado, visto que foi possível aprender *frameworks* ainda desconhecidos pelo autor deste trabalho.

Como trabalhos futuros, sugere-se a implementação de funcionalidades adicionais como, por exemplo, uma *checklist* para o usuário marcar se já ministrou um determinado medicamento em um determinado horário para um residente, envio de alertas quando um medicamento não for ministrado no horário correto, um controle de estoque para os medicamentos, armazenamento de arquivos e um serviço de recuperação de senhas para os usuários.

REFERÊNCIAS

ALLAIRE, Jeremy. **Requirements for rich internet applications**. 2002. Disponível em: http://download.macromedia.com/pub/flash/whitepapers/richclient.pdf. Acesso em: 11 mar. 2019.

ALVARENGA, Darlan; BRITO, Carlos. **1 em cada 4 brasileiros terá mais de 65 anos em 2060, aponta IBGE**. 2018. G1 - São Paulo e Rio de Janeiro. Disponível em: https://g1.globo.com/economia/noticia/2018/07/25/1-em-cada-4-brasileiros-tera-mais-de-65-anos-em-2060-aponta-ibge.ghtml. Acesso em: 20 mar. 2019.

BERNARDI, Mario Luca; DI LUCCA, Giuseppe Antonio; DISTANTE, Damiano. **Model-Driven fast prototyping of RIAs**: from conceptual models to running applications. IEEE, 2014, p. 250-258.

BOZZON, Alessandro; COMAI, Sara; FRATERNALI, Piero; CARUGHI, Giovanni T. **Capturing RIA concepts in a web modeling language**. In: 15th international Conference on World Wide Web, ACM, 2006, p. 907-908.

BUSCH, Marianne; KOCH, Nora. **Rich internet applications**: state-of-the-art. Technical Report 0902, 2009, p. 1-18.

DINO. **Número de idosos no Brasil cresceu 50% em uma década, segundo IBGE**. Disponível em: https://www.terra.com.br/noticias/dino/numero-de-idosos-no-brasil-cresceu-50-em-uma-decada-segundo-ibge,6427cac70c638ddd25efe9c43fb7d977r5spkpo1.html. Acesso em: 18 mar. 2019.

FRATERNALI, Piero; ROSSI, Gustavo; SANCHEZ-FIGUEROA, Fernando. **Rich internet applications**. IEEE Internet Computing, may/June, 2010, p. 9-14.

LIPSCHITZ, D.A. Screening for Nutritional Status in the Elderly. **Primary Care**, v. 21, n. 1, mar.1994, p. 55-67.

MACHADO, Leonardo, BERNARDO FILHO, Orlando; RIBEIRO, João. UWE-R: an extension to a web engineering methodology for rich internet applications. **WSEAS Trans. Info. Sci. and App.**, v. 6, n. 4, abr. 2009, p. 601-610.

MELIÁ, Santiago; GÓMEZ, Jaime; PÉREZ, Sandy; DÍAZ, Oscar. **A model-driven development for GWT-based rich internet applications with OOH4RIA**. In: 8th International Conference on Web Engineering (ICWE'08), IEEE, 2008, p. 13-23.

ORGANIZAÇÃO MUNDIAL DA SAÚDE. **Obesity**: preventing and managing the global epidemic. Report of a WHO Consultation (WHO Technical Report Series 894). 2000, p. 9. Disponível em: https://www.who.int/nutrition/publications/obesity/WHO_TRS_894/en. Acesso em: 05 mai. 2020.

PARADELLA, Rodrigo. **Número de idosos cresce 18% em 5 anos e ultrapassa 30 milhões em 2017**. IBGE: Estatísticas Sociais. Disponível em: https://agenciadenoticias.ibge.gov.br/agencia-noticias/2012-agencia-de-noticias/noticias/20980-numero-de-idosos-cresce-18-em-5-anos-e-ultrapassa-30-milhoes-em-2017. Acesso em: 18 mar. 2019.

VALVERDE, Francisco; PASTOR, Oscar. **Facing the technological challenges of web 2. 0**: a RIA model-driven engineering approach. In Proc. of the International Conference on WISE, WISE 2009, p. 1-15.

VILARDAGA, Vicente; CAVICCHIOLI, Giorgia. **O abandono dos idosos no Brasil**. Disponível em: https://istoe.com.br/o-abandono-dos-idosos-no-brasil. Acesso em: 18 mar. 2019.