
**Universidade Federal de Ouro Preto Instituto de Ciências Exatas e Aplicadas Departamento de Computação e Sistemas**

# ODONTO SISTEM – SISTEMA WEB DE GERENCIAMENTO ODONTOCLÍNICO

**Gabriele Carla Cordeiro Araújo**

## TRABALHO DE CONCLUSÃO DE CURSO

ORIENTAÇÃO:

Fernando Bernardes de Oliveira

**Agosto, 2023**

**João Monlevade – MG**

**Gabriele Carla Cordeiro Araújo**

# ODONTO SISTEM - SISTEMA WEB DE GERENCIAMENTO ODONTOCLÍNICO

Orientador: Fernando Bernardes de Oliveira

Monografia apresentada ao curso de Sistemas de Informação do Instituto de Ciências Exatas e Aplicadas, da Universidade Federal de Ouro Preto, como requisito parcial para aprovação na Disciplina “Trabalho de Conclusão de Curso II”.

**Universidade Federal de Ouro Preto**

**João Monlevade - MG**

**Agosto, 2023**

SISBIN - SISTEMA DE BIBLIOTECAS E INFORMAÇÃO

A663o Araujo, Gabriele Carla Cordeiro.
Odonto Sistem [manuscrito]: sistema web de gerenciamento odontoclínico. / Gabriele Carla Cordeiro Araujo. - 2023.
129 f.: il.: color., tab..

Orientador: Prof. Dr. Fernando Bernardes de Oliveira.
Monografia (Bacharelado). Universidade Federal de Ouro Preto. Instituto de Ciências Exatas e Aplicadas. Graduação em Sistemas de Informação .

1. Administração de empresas. 2. Aplicações Web. 3. Cadastros - Pacientes. 4. Clínicas dentárias. 5. Sistemas de informação gerencial. I. Oliveira, Fernando Bernardes de. II. Universidade Federal de Ouro Preto. III. Título.

CDU 004.775

Bibliotecário(a) Responsável: Flavia Reis - CRB6-2431

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE OURO PRETO
REITORIA
INSTITUTO DE CIENCIAS EXATAS E APLICADAS
DEPARTAMENTO DE COMPUTACAO E SISTEMAS

**FOLHA DE APROVAÇÃO**

**Gabriele Carla Cordeiro Araújo**

**Odonto Sistem - Sistema *web* de gerenciamento odontoclínico**

Monografia apresentada ao Curso de Sistemas de Informação da Universidade Federal
de Ouro Preto como requisito parcial para obtenção do titulo de Bacharel em Sistemas de Informação

Aprovada em 28 de agosto de 2023

Membros da banca

Prof. Dr. Fernando Bernardes de Oliveira - Orientador (Universidade Federal de Ouro Preto)
Profª. Drª. Gilda Aparecida de Assis - Avaliadora (Universidade Federal de Ouro Preto)
Prof. Dr. Rafael Frederico Alexandre - Avaliador (Universidade Federal de Ouro Preto)

Fernando Bernardes de Oliveira, orientador do trabalho, aprovou a versão final e autorizou seu depósito na Biblioteca Digital de Trabalhos de Conclusão de Curso da UFOP em 26/09/2023

Documento assinado eletronicamente por **Fernando Bernardes de Oliveira**, **PROFESSOR DE MAGISTERIO SUPERIOR**, em 26/09/2023, às 19:06, 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 http://sei.ufop.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0 , informando o código verificador **0595394** e o código CRC **03D85AAB**.



R. Diogo de Vasconcelos, 122, - Bairro Pilar Ouro Preto/MG, CEP 35402-163
Telefone: (31)3808-0819 - www.ufop.br

*Dedico este trabalho de dedicação e esforço aos meus pais, noivo, familiares e amigos que sempre estiveram ao meu lado, apoiando e incentivando meu crescimento acadêmico. Vocês são minha força e inspiração.*

## AGRADECIMENTOS

Gostaria de expressar meus sinceros agradecimentos a todos que contribuíram para a conclusão desta monografia. Em primeiro lugar, agradeço aos meus pais, cujo amor e apoio constante foram fundamentais em todos os momentos da minha jornada acadêmica. Vocês são minha base e fonte de inspiração, e sou imensamente grata pela confiança que depositaram em mim. Ao meu noivo, que esteve ao meu lado durante todo o processo, agradeço por compreender os momentos de dedicação intensa e por sempre me encorajar a dar o meu melhor. Sua paciência, compreensão e incentivo foram essenciais para que eu me mantivesse motivada até o fim.

Não posso deixar de mencionar meus familiares e amigos, que sempre estiveram presentes, oferecendo palavras de encorajamento e suporte emocional. Sou grata pela maneira como vocês me impulsionaram para alcançar meus objetivos.

Também gostaria de expressar minha gratidão ao meu orientador, Dr. Fernando Bernardes, que dedicou seu tempo e conhecimento para me guiar no desenvolvimento deste trabalho. Sua orientação construtiva foi fundamental para moldar minha visão e aprimorar minhas habilidades. Sou imensamente grata pela confiança depositada em mim e pela oportunidade de aprender com você.

Quero expressar meu profundo agradecimento à Aliare, a empresa na qual tenho a honra de trabalhar. Por meio das oportunidades proporcionadas pelo trabalho com desenvolvimento web e mobile, a Aliare tem sido fundamental em minha jornada profissional, permitindo-me crescer e aprimorar minhas habilidades de forma significativa. Os ensinamentos recebidos, o ambiente colaborativo e o suporte da equipe têm sido inestimáveis para meu desenvolvimento como desenvolvedora. Através da Aliare, pude realizar meu sonho de seguir nessa área e me tornar a profissional que sempre almejei ser. Obrigada por acreditar em mim e por me oferecer as ferramentas e oportunidades para alcançar meu potencial.

Este trabalho não seria possível sem a colaboração e o suporte de todos vocês. A todos os mencionados e aqueles que não foram citados individualmente, meu mais profundo agradecimento.

*"A única maneira de fazer um ótimo trabalho é amar o que você faz."*

*(Steve Jobs)*

## RESUMO

Ao observar os processos diários realizados em clínicas odontológicas, tornou-se evidente a necessidade de um sistema web eficiente para otimizar as atividades administrativas. Embora essas tarefas sejam simples, consomem uma quantidade significativa de tempo, assim, a adoção de um software que auxilie nos processos administrativos da clínica se torna crucial. Diante desse contexto, o presente trabalho tem como objetivo propor uma aplicação web abrangente para a gestão administrativa completa da clínica. Dentre as funcionalidades implementadas, destacam-se o agendamento de consultas, a atualização de prontuários contendo informações relevantes, como procedimentos realizados, anamnese e observações. Além disso, o sistema permite o cadastro de pacientes, clínicas e dentistas, proporcionando uma visão organizada e abrangente de todas as informações necessárias para o funcionamento adequado da clínica odontológica. A implementação do sistema web foi guiada pela simplicidade e facilidade de uso, com o intuito de atender às demandas específicas desse ambiente. Utilizaram-se tecnologias modernas e adequadas para o desenvolvimento web, resultando em uma interface amigável e intuitiva para os usuários. Após a conclusão da implementação do sistema, foi realizado um teste piloto envolvendo um grupo de usuários, composto por profissionais da área de odontologia que se voluntariaram para avaliar as funcionalidades do software. Durante essa fase, foram coletados feedbacks valiosos dos colaboradores e com base nos resultados obtidos, foi constatado que o sistema atende efetivamente às demandas dos usuários e é capaz de complementar as metodologias utilizadas pelos profissionais, contribuindo para uma gestão mais eficiente e otimizada.



**Palavras-chave:** clínicas odontológicas; sistema web; gestão administrativa; otimização; cadastro de pacientes.

## ABSTRACT

When observing the daily processes carried out in dental clinics, the need for an efficient web-based system to optimize administrative activities became evidente, although these tasks are simple, they consume a significant amount of time, making the adoption of software that assists in clinic administrative processes crucial. In this context, the aim of this work is to develop a comprehensive web application for complete administrative management of the clinic. Among the implemented functionalities, notable features include appointment scheduling, updating records with relevant information such as performed procedures, medical history, and observations. Additionally, the system allows for the registration of patients, clinics, and dentists, providing an organized and comprehensive view of all necessary information for the proper functioning of the dental clinic. The implementation of the web-based system was guided by simplicity and user-friendliness, intending to meet the specific demands of this environment. Modern and appropriate web development technologies were employed, resulting in a user-friendly and intuitive interface. Following the completion of system implementation, a pilot test was conducted involving a group of users, comprising dental professionals who volunteered to evaluate the software's functionalities. During this phase, valuable feedback was collected from the participants, and based on the obtained results, it was confirmed that the system effectively meets user demands and complements the methodologies used by professionals, contributing to more efficient and optimized management.



**Keywords:** dental clinics; web system; administrative management; optimization; patient registration.

# LISTA DE FIGURAS

Figura 1 - DER – Usuários, Dentistas, Pacientes, Clínicas, Prontuários e Agenda ..... 32
Figura 2 - DER – Roles, Permissions ..... 34
Figura 3 - Trecho de código da tela do Dashboard – Desempenho Geral ..... 35
Figura 4 - Trecho de código arquivo JavaScript – Desempenho Geral ..... 36
Figura 5 - Trecho de código arquivo JavaScript – Desempenho Geral ..... 37
Figura 6 - Trecho de código arquivo JavaScript – Desempenho Geral ..... 39
Figura 7 - Trecho de código arquivo JavaScript - Pacientes por estado ..... 42
Figura 8 - Trecho de código arquivo JavaScript - Pacientes por estado ..... 42
Figura 9 - Trecho de código da tela do Dashboard – Pacientes por estado ..... 44
Figura 10 - Trecho de código da tela do Dashboard – Pacientes por estado ..... 45
Figura 11 - Home Controller ..... 46
Figura 12 - Tela Dashboard ..... 47
Figura 13 – Cabeçalho - Tela Paciente ..... 48
Figura 14 – Busca - Tela paciente ..... 49
Figura 15 – Corpo tabela – Tela Paciente ..... 51
Figura 16 – Tela Pacientes ..... 53
Figura 17 – Tela de visualização – Paciente ..... 54
Figura 18 – Tela create – Paciente ..... 56
Figura 19 – Corpo do formulário – Paciente ..... 57
Figura 20 – Controller de criação ..... 60
Figura 21 – Controller de edição ..... 62
Figura 22 – JavaScript de exclusão ..... 64
Figura 23 – JavaScript de exclusão ..... 65
Figura 24 – Model Patients ..... 66
Figura 25 – Rotas web.php ..... 67
Figura 26 – HTML de criação do calendário ..... 69
Figura 27 – Calendário – JavaScript ..... 70
Figura 28 – Calendário – JavaScript ..... 72
Figura 29 – Calendário – JavaScript ..... 74
Figura 30 - Tela de Agenda ..... 75
Figura 31 – Calendário – Modal Agendamento ..... 76
Figura 32 – Calendário – Modal Agendamento ..... 77
Figura 33 – Calendário – Modal Agendamento ..... 78
Figura 34 – Calendário – Modal Agendamento ..... 79
Figura 35 – Calendário – Modal Agendamento ..... 80

Figura 36 - Modal Agendamento ..... 81
Figura 37 – Calendário – Agendamento JavaScript ..... 82
Figura 38 – Calendário – Agendamento JavaScript ..... 84
Figura 39 – Calendário – Exclusão JavaScript ..... 86
Figura 40 – Calendário – Evento de Click ..... 87
Figure 41 – Calendário – Evento de Click ..... 89
Figura 42 – Calendário – Evento de Drop e Resize ..... 91
Figura 43 – Calendário – Evento de Drop e Resize ..... 92
Figura 44 – Validação Data Primeira Consulta ..... 94
Figura 45 – Validação Data Primeira Consulta ..... 95
Figura 46 – Validação Data Nascimento ..... 96
Figura 47 – Validação Idade Dentista ..... 97
Figura 48 – Validação CEP e CPF ..... 98
Figura 49 – Validação CRO ..... 99
Figura 50 – Validação Data Prontuário ..... 100
Figura 51 – Validação Data Prontuário ..... 101
Figura 52 – Validação Obrigatoriedade ..... 103
Figura 53 – Tela de login ..... 109
Figura 54 – Tela de registro ..... 110
Figura 55 – Tela cadastro pacientes ..... 111
Figura 56 – Tela edição pacientes ..... 112
Figura 57 – Tela visualização de pacientes ..... 113
Figura 58 – Tela de Prontuários ..... 114
Figura 59 – Tela de Clínicas ..... 115
Figura 60 – Tela de Dentistas ..... 116
Figura 61 – Tela Perfil de Usuário ..... 117
Figura 62 – Tela Gestão de Usuários ..... 118
Figura 63 - Diagramas de Casos de Uso – Agenda ..... 122
Figura 64 - Diagramas de Casos de Uso - Pacientes ..... 123
Figura 65 - Diagramas de Casos de Uso - Perfil de Usuário ..... 123
Figura 66 - Diagramas de Casos de Uso – Dentista ..... 124
Figura 67 - Diagramas de Casos de Uso - Clínica ..... 125
Figura 68 - Diagramas de Casos de Uso - Prontuário ..... 125
Figura 69 - Diagramas de Casos de Uso - Gestão de Usuários ..... 126
Figura 70 - Diagramas de Casos de Uso - Painel Inicial ..... 126
Figura 71 - Diagrama EER – Completo ..... 127

# LISTA DE TABELAS

Tabela 1 - Requisitos funcionais ..... 26
Tabela 2 – Requisitos não funcionais..... 27

## SUMÁRIO


**1. INTRODUÇÃO ..... 14**

**1.1. PROBLEMA ..... 15**

**1.2. OBJETIVOS ..... 16**

**1.3. METODOLOGIA ..... 17**

**1.4. ESTRUTURA DO TRABALHO ..... 18**

**2. TECNOLOGIAS E FUNDAMENTOS PARA O DESENVOLVIMENTO DO PROJETO ..... 19**

**2.1. FRAMEWORK ..... 19**

**2.2. PADRÃO DE PROJETO ADOTADO ..... 20**

**2.3. BANCO DE DADOS ..... 21**

**2.4. REFERÊNCIAS PARA O DESENVOLVIMENTO DO PROJETO ..... 22**

**3. DESENVOLVIMENTO DO SISTEMA: ETAPAS E RECURSOS ..... 24**

**3.1. LEVANTAMENTO DE REQUISITOS ..... 24**

3.1.1. Identificação das necessidades do usuário ..... 24

3.1.2. Definição de requisitos funcionais e não funcionais ..... 25

3.1.3. Identificação dos atores ..... 27

3.1.4. Definição dos casos de uso ..... 28

**3.2. DESIGN DE INTERFACE ..... 29**

**3.3. DESENVOLVIMENTO DE FUNCIONALIDADES ..... 30**

3.3.1. Modelagem do banco de dados ..... 31

3.3.2. Tela principal ..... 34

3.3.3. Telas de cadastro ..... 47

3.3.3.1. Tela inicial de pacientes ..... 48

3.3.3.2. Visualização de registro ..... 53

3.3.3.3. Cadastro ou edição de registro ..... 56

3.3.3.4. Exclusão de um registro ..... 63

3.3.3.5. Model e Rotas CRUD ..... 66

3.3.4. Calendário ..... 68
3.3.4.1. Criação do calendário ..... 69
3.3.4.2. Select ..... 71
3.3.4.3. Modal para agendamento ..... 75
3.3.4.4. Cadastrar ou Editar um evento ..... 81
3.3.4.5. Exclusão de um evento ..... 86
3.3.4.6. Event Click ..... 87
3.3.4.7. Event Drop e Resize ..... 90
**3.4. VALIDAÇÕES ..... 93**
3.4.1. Datas ..... 94
3.4.2. CEP, CPF e CRO ..... 98
3.4.3. Datas do Prontuário ..... 100
3.4.4. Campos obrigatórios ..... 102
**3.5. TESTES ..... 104**
3.5.1. Abordagem utilizada ..... 104
3.5.2. Cenários simulados ..... 105
**3.6. RESOLUÇÃO DE PROBLEMAS E AJUSTES ..... 106**
**4. ANÁLISE DOS RESULTADOS E AVALIAÇÃO DO SISTEMA ..... 108**
**4.1. TELAS DA APLICAÇÃO ..... 108**
4.1.1. Tela de Login/Registrar ..... 108
4.1.1.1. Login ..... 108
4.1.1.2. Registrar ..... 109
4.1.2. Tela de Pacientes ..... 110
4.1.2.1. Cadastro ..... 110
4.1.2.2. Edição ..... 111
4.1.2.3. Visualização ..... 112
4.1.3. Tela de Prontuários ..... 113
4.1.4. Telas de Gerenciamento ..... 114
4.1.4.1. Clínicas ..... 114
4.1.4.2. Dentistas ..... 115

4.1.5. Telas de Configurações ..... 116
4.1.5.1. Perfil de usuário ..... 116
4.1.5.2. Gestão de usuários ..... 117
**4.2. CONSIDERAÇÕES FINAIS ..... 118**
**5. CONCLUSÕES ..... 119**
5.1. Trabalhos futuros ..... 120
**REFERÊNCIAS ..... 121**
**APÊNDICES ..... 122**
**APÊNDICE A – DIAGRAMAS DE CASOS DE USO ..... 122**
**APÊNDICE B – DIAGRAMA DE ENTIDADE E RELACIONAMENTO ..... 127**

# 1. Introdução

A introdução às consultas em clínicas odontológicas tem sido um processo essencial para a saúde bucal da população ao longo dos anos. Antigamente, o agendamento de consultas era feito de maneira manual, envolvendo fichas de papel, registros em agendas físicas e, por vezes, até mesmo telefonemas. Esse método tradicional de marcação de consultas frequentemente resultava em ineficiências, tais como perda de informações, falta de organização e dificuldade de gerenciamento do fluxo de pacientes.

Na rotina de uma clínica odontológica, uma série de processos são realizados diariamente, desde o agendamento de consultas até o registro e acompanhamento dos tratamentos dos pacientes. Segundo Hartmann e Cantarelli (2021), a tecnologia desempenha um papel fundamental na otimização desses processos, o que pode levar a uma maior eficiência no atendimento aos pacientes.

No entanto, com os avanços tecnológicos e a crescente digitalização dos processos, surgiram soluções inovadoras para otimizar o agendamento de consultas em clínicas odontológicas. A adoção de sistemas web para a gestão administrativa trouxe benefícios significativos, permitindo uma marcação mais eficiente, redução de erros e uma melhor experiência tanto para os profissionais de saúde quanto para os pacientes.

Com base no artigo encontrado no site SimDoctor (2018), é ressaltado que um *software* odontológico proporciona diversas vantagens para a prática odontológica. Uma das principais vantagens é a possibilidade de realizar o agendamento e cancelamento de consultas de forma *online*, o que contribui para melhorar a experiência dos pacientes. Essa funcionalidade proporciona uma maior comodidade aos pacientes, permitindo que agendem suas consultas de forma mais rápida e conveniente, além de facilitar a gestão dos horários disponíveis para os profissionais.

Além disso, o *software* odontológico desempenha um papel importante na gestão administrativa da clínica, integrando diversas funções necessárias para o atendimento dos pacientes e a gestão financeira. Essas funcionalidades

proporcionam aos dentistas uma maior eficiência em seu cotidiano, permitindo uma melhor organização dos serviços prestados.

Através de um *software* odontológico, é possível acessar rapidamente as informações dos pacientes, agendar consultas, registrar prontuários e gerenciar o histórico clínico, tudo de forma segura e organizada. Assim, a transição do agendamento manual para um sistema web na gestão das clínicas odontológicas representa um avanço significativo, proporcionando benefícios tanto para os profissionais de saúde, otimizando suas tarefas, quanto para os pacientes, que desfrutam de uma experiência mais satisfatória em suas consultas.

## 1.1. Problema

As clínicas odontológicas são ambientes que demandam uma gestão eficiente para garantir um atendimento de qualidade aos pacientes. Nesse sentido, a utilização de sistemas de gestão específicos para o setor tem se mostrado uma prática cada vez mais adotada. A implementação de sistemas de gestão para clínicas odontológicas traz benefícios significativos, como melhorias nas tomadas de decisões, controle de informações, atendimento aos clientes e avanços no processo produtivo (SimDoctor, 2018).

Hartmann e Cantarelli (2021) afirmam que a adoção de tecnologias de informação, como sistemas de gestão, amplia a produtividade e organização das clínicas. Isso facilita o acesso dos dentistas a informações sobre horários e clientes, reduzindo a carga de trabalho dos secretários e evitando perda de horários devido a esquecimentos. O artigo destaca que a adoção de sistemas de gestão permite uma melhor organização dos processos administrativos, o que resulta em uma maior eficiência operacional e na redução de erros. Além disso, a utilização dessas ferramentas possibilita o controle adequado dos prontuários dos pacientes, o agendamento eficiente de consultas, a gestão do estoque de materiais odontológicos e a melhoria da comunicação entre os membros da equipe.

Outro ponto relevante abordado no estudo é o impacto positivo que a implementação de sistemas de gestão tem na experiência do paciente. Com uma gestão mais eficiente, as clínicas podem oferecer um atendimento mais ágil,

personalizado e de qualidade, o que contribui para a satisfação dos pacientes e para o fortalecimento do relacionamento com eles.

Ao adotar um *software* de gestão odontológica, as clínicas têm a oportunidade de aprimorar seus processos internos, reduzindo erros, aumentando a eficiência operacional e proporcionando um atendimento de maior qualidade aos pacientes. A melhoria na gestão administrativa resulta em benefícios tangíveis, como a redução de custos, a maximização do tempo e recursos, além do fortalecimento da imagem da clínica perante os pacientes.

Portanto, com base nas evidências apresentadas no estudo de Hartmann e Cantarelli (2021), fica evidente que a adoção de sistemas de gestão para clínicas odontológicas é uma estratégia importante para aprimorar a administração das clínicas e oferecer um serviço de excelência. Essas ferramentas permitem uma gestão mais eficiente e integrada, resultando em melhorias significativas na qualidade dos processos, na experiência do paciente e na competitividade das clínicas no mercado odontológico.

## 1.2. Objetivos

O objetivo geral deste trabalho é desenvolver uma plataforma *web* para gerenciamento de clínicas odontológicas. Através dessa solução, as clínicas terão acesso a uma variedade de recursos, como agendamento de consultas, controle de prontuários, gestão de pacientes e outros recursos administrativos que visam otimizar a organização e o desempenho das clínicas odontológicas.

Esse trabalho possui os seguintes objetivos específicos:

- Modelar e implementar a aplicação *web*: o primeiro objetivo é criar uma estrutura sólida e funcional para a plataforma, abrangendo as funcionalidades necessárias para o gerenciamento eficiente de uma clínica odontológica. Isso inclui o desenvolvimento de interfaces intuitivas e amigáveis para facilitar a utilização pelos profissionais de saúde.
- Avaliar a aplicação por meio da experiência dos usuários: para garantir a eficiência e a usabilidade da plataforma, é essencial realizar testes com

usuários reais, como dentistas, assistentes administrativos e outros profissionais envolvidos na rotina da clínica odontológica. Com base no *feedback* desses usuários, será possível identificar possíveis melhorias e ajustes necessários para aprimorar a experiência de uso.

Ao cumprir esses objetivos específicos, o resultado esperado é uma plataforma *web* funcional para o gerenciamento eficiente de clínicas odontológicas.

## 1.3. Metodologia

Para a execução deste trabalho, foram estabelecidos os seguintes passos:

1. Revisão da literatura: foi realizado um estudo abrangente das possíveis tecnologias para o desenvolvimento da aplicação, bem como uma avaliação de aplicações e trabalhos correlatos. Esse processo permitiu identificar pontos de melhoria que o software poderia oferecer.

2. Levantamento de requisitos: foram identificadas as principais necessidades cotidianas das clínicas odontológicas, a fim de estabelecer os requisitos necessários para o melhor gerenciamento da clínica.

3. Definição da documentação do *software*: uma documentação detalhada do *software* foi estabelecida, contendo todos os requisitos, etapas e cronograma do processo de implementação.

4. Modelagem da aplicação: foi realizada a modelagem e prototipação do *software* de acordo com os requisitos estabelecidos, levando em consideração as necessidades dos usuários.

5. Implementação: o sistema foi desenvolvido, levando em conta os requisitos pré-estabelecidos e as necessidades dos usuários, mantendo fidelidade ao protótipo estabelecido.

6. Testes: foram planejados e realizados com usuários da área, a fim de obter *feedback* e destacar possíveis melhorias.

7. Análise: os testes executados foram analisados, discutindo-se as melhorias identificadas a partir dos resultados obtidos. Essas melhorias foram implementadas com o objetivo de aprimorar a experiência do usuário.

## 1.4. Estrutura do trabalho

A estrutura deste trabalho é organizada da seguinte forma: no Capítulo 2, abordam-se os conceitos relacionados ao projeto, incluindo as tecnologias utilizadas, como o *framework* escolhido, o padrão de projeto adotado, o banco de dados e os sistemas de referência. No Capítulo 3, descreve-se o processo de desenvolvimento do sistema, com detalhes sobre etapas, recursos e criação das telas. Também são abordados os resultados dos testes com profissionais, avaliando a usabilidade, eficiência e adequação do sistema às necessidades dos usuários. O Capítulo 4 descreve os resultados obtidos durante o processo de desenvolvimento do *software*, destacando as principais telas do sistema, suas funcionalidades e interfaces. Por fim, o Capítulo 5 traz as considerações finais e propostas para trabalhos futuros.

# 2. Tecnologias e Fundamentos para o Desenvolvimento do Projeto

Este capítulo aborda os conceitos e tecnologias utilizadas no desenvolvimento do projeto, fornecendo uma base sólida de conhecimento sobre as bases tecnológicas adotadas. Na seção 2.1, será abordada a escolha da linguagem de programação e uma introdução sobre o *framework*, o padrão de projeto, o banco de dados e as *APIs* utilizadas. Em seguida, na seção 2.2, serão discutidas as vantagens e desvantagens do *framework* escolhido, bem como seus conceitos e exemplos de uso. Na seção 2.3, serão apresentados os benefícios e a aplicabilidade do padrão de projeto adotado, juntamente com exemplos práticos de seu uso. A seção 2.4 abordará o tipo de banco de dados utilizado, a modelagem realizada e as estratégias de segurança e *backup* implementadas. Por fim, na seção 2.5, serão apresentadas as aplicações ou sistemas utilizados como referência, bem como uma análise das funcionalidades e características desses sistemas, que serviram de inspiração e aprendizado para o desenvolvimento do projeto.

## 2.1. Framework

O *Laravel* é um *framework* de desenvolvimento *web* em *PHP* que ganhou grande popularidade nos últimos anos. De acordo com um estudo comparativo realizado por LAAZIRI et al. (2019), que analisou os *frameworks PHP Laravel* e *Symfony*, o *Laravel* é conhecido por sua elegância, flexibilidade e pelos recursos avançados que oferece aos desenvolvedores. O *framework* segue o padrão de arquitetura *MVC*, permitindo uma separação clara das preocupações em um aplicativo *web*, o que facilita o desenvolvimento, a manutenção e a escalabilidade. Essa abordagem é destacada como uma das principais vantagens do *Laravel*, conforme destacado no artigo de referência.

Segundo Laaziri, El Bouhdidi e El Mohajir (2019), em seu estudo comparativo sobre frameworks *PHP* para o desenvolvimento de aplicações *web*, o *Laravel* foi destacado como uma excelente opção para o desenvolvimento rápido de aplicativos em grande escala, mesmo para desenvolvedores com menos experiência. Os autores também mencionam que o Laravel é um framework que oferece recursos úteis, como roteamento flexível, migrações de banco de dados, autenticação de usuário, filas de trabalho e notificações. Além disso, a comunidade ativa de

desenvolvedores e os recursos de suporte, como fóruns[1], tutoriais e documentação detalhada, são aspectos destacados pelos autores. No contexto de desempenho, o *Laravel* superou outros frameworks *MVC* em testes de avaliação, como taxas de solicitações por segundo, eficiência de uso de memória, tempo de resposta e número mínimo de arquivos necessários.

No entanto, assim como qualquer tecnologia, o *Laravel* também possui algumas desvantagens. De acordo com um estudo comparativo entre os *frameworks .NET* e *Laravel* realizado por Richard (2022), é importante considerar que o aprendizado inicial do *Laravel* pode ser um tanto desafiador, especialmente para desenvolvedores inexperientes em *PHP* ou *frameworks* MVC. O autor destaca que o *Laravel* pode não ser a melhor escolha se a equipe de desenvolvimento não estiver familiarizada com a linguagem *PHP*. Além disso, é importante ressaltar que o *Laravel* é um *framework* de alto nível, o que pode torná-lo menos adequado para projetos de menor escala ou com requisitos de desempenho extremamente exigentes.

Sendo assim, através das pesquisas realizadas e dos estudos comparativos apresentados, fica evidente que o *Laravel* é uma opção sólida e confiável para o desenvolvimento de aplicações *web* em *PHP*. Sua elegância, flexibilidade e recursos avançados proporcionam uma experiência de desenvolvimento aprimorada, permitindo uma separação clara das preocupações através do padrão de arquitetura *MVC*.

## 2.2. Padrão de Projeto Adotado

O padrão de projeto adotado no projeto foi o *MVC (Model-View-Controller).* O *MVC* é um padrão arquitetural amplamente utilizado no desenvolvimento de *software*, incluindo aplicações *web*. Ele tem como objetivo separar as responsabilidades em três componentes principais: o Modelo (*Model*), a Visão (*View*) e o Controlador (*Controller*). Uma das principais vantagens do padrão *MVC* é a clara separação de

[1] **Laravel Forums (Laracasts):** Fórum oficial do Laravel hospedado no Laracasts. Acesse aqui.
**Laravel.io:** Comunidade dedicada ao Laravel. Acesse aqui.
**r/laravel (Reddit):** Subreddit popular para discussões sobre o Laravel no Reddit. Acesse aqui.

preocupações. O Modelo é responsável pela lógica de negócio e manipulação dos dados, a Visão lida com a apresentação e exibição dos dados ao usuário e o Controlador atua como intermediário entre o Modelo e a Visão, gerenciando as interações e ações do usuário.

De acordo com Luciano e Alves (2017), o padrão de arquitetura *MVC (Model-view-controller)* facilita a manutenção e o desenvolvimento do sistema, uma vez que cada componente possui um papel bem definido. Alterações no Modelo não afetam a Visão e vice-versa, o que permite realizar modificações específicas sem afetar todo o sistema. Além disso, o padrão *MVC* facilita a reutilização de código. O Modelo pode ser reutilizado em diferentes aplicações, pois não está diretamente acoplado à interface do usuário. Da mesma forma, é possível ter diferentes Visões para o mesmo Modelo, adaptando a apresentação dos dados conforme as necessidades específicas.

A aplicabilidade do padrão *MVC* é ampla, sendo adequado para projetos de diferentes tamanhos e complexidades (Lemos et al., 2013). Ele é particularmente útil em sistemas nos quais a lógica de negócio é separada da interface do usuário, permitindo uma melhor organização e manutenção do código (Luciano & Alves, 2017). O *MVC* também facilita a colaboração entre equipes de desenvolvimento, uma vez que as responsabilidades são claramente definidas e podem ser trabalhadas em paralelo.

No contexto do projeto em questão, a adoção do padrão *MVC* trouxe benefícios como a organização do código, a facilidade de manutenção e a escalabilidade do sistema. Ele permitiu separar as funcionalidades de forma coerente e modular, tornando o desenvolvimento mais eficiente e aprimorando a qualidade do *software*.

## 2.3. Banco de Dados

O banco de dados utilizado no projeto foi o *MySQL*, um sistema de gerenciamento de banco de dados relacional amplamente utilizado na indústria de desenvolvimento de *software*. O *MySQL* oferece uma combinação de confiabilidade, desempenho e escalabilidade, tornando-o uma escolha popular para aplicações *web*

(BENTO, 2021, p.54). Uma das principais vantagens do *MySQL* é a sua ampla compatibilidade com várias linguagens de programação, incluindo o *PHP* utilizado neste projeto. Isso facilita a integração do banco de dados com a lógica de negócio do sistema.

O *MySQL Workbench* 8.0 foi utilizado como ferramenta de modelagem e administração do banco de dados. De acordo com o estudo realizado por INAN, Dedi Iskandar, e JUITA, Ratna (2011), a ferramenta *MySQL Workbench* fornece uma interface gráfica intuitiva para projetar esquemas de banco de dados, criar e modificar tabelas, executar consultas *SQL* e realizar tarefas de manutenção.

A escolha do *MySQL* e do *MySQL Workbench* 8.0 se deu pela sua reputação de confiabilidade, ampla adoção na indústria e disponibilidade de recursos avançados. Essas ferramentas foram essenciais para o armazenamento e gerenciamento eficiente dos dados do sistema, contribuindo para o sucesso do projeto.

## 2.4. Referências para o Desenvolvimento do Projeto

Durante o desenvolvimento do projeto, foram utilizados dois sistemas como referência para obter insights e inspiração. O primeiro sistema é o *Codental*[2], que se destaca por sua funcionalidade de agendamento online, facilitando o processo de marcação de consultas. Além disso, o *Codental* oferece um prontuário digital completo, substituindo o uso de fichas de papel e proporcionando maior organização e praticidade. Seus recursos são projetados para tornar a clínica mais competitiva, combinando funcionalidade e uma interface bonita e fácil de usar.

O segundo sistema de referência é o Controle Odonto[3], que compartilha a ideia de gestão de clínicas odontológicas, controle de pacientes e agendamentos de acordo com a disponibilidade dos dentistas. O Controle Odonto se destaca por seu *dashboard* atraente, que oferece uma visão geral das atividades da clínica. Além disso, assim como o *Codental*, o Controle Odonto também prioriza a simplicidade e a

[2] Codental | Software Odontológico Amado Pelos Dentistas

[3] Software Odontológico Mais Completo - ControleODONTO

intuitividade em seu sistema, proporcionando uma experiência agradável para os usuários.

Ambos os sistemas foram utilizados como fonte de referência para identificar boas práticas, recursos relevantes e abordagens eficientes na criação do sistema proposto. A análise desses sistemas contribuiu para o desenvolvimento de um sistema completo, que atende às necessidades de gestão de uma clínica odontológica, ao mesmo tempo em que oferece uma interface intuitiva e agradável aos usuários.

# 3. Desenvolvimento do Sistema: Etapas e Recursos

O Capítulo 3 aborda o detalhamento do processo de desenvolvimento do sistema, com ênfase nas etapas e recursos utilizados para a criação das telas. Neste capítulo, serão apresentados os procedimentos e técnicas empregados para garantir a funcionalidade e usabilidade adequada do sistema. Além disso, um tópico importante a ser explorado é a validação de campos, que desempenham um papel crucial na integridade dos dados e na garantia da consistência das informações inseridas pelos usuários. Através deste capítulo, busca-se fornecer uma visão abrangente de todo o processo de desenvolvimento, desde a criação das telas até a implementação das validações necessárias para assegurar a qualidade e eficiência do sistema.

## 3.1. Levantamento de requisitos

A fim de verificar as principais funcionalidades do *software*, o levantamento de requisitos desempenhou um papel crucial no processo de desenvolvimento do sistema. Por meio desse processo, foram identificadas e definidas as necessidades e expectativas dos usuários envolvidos no projeto. Neste tópico, serão apresentados em detalhes os passos realizados no levantamento de requisitos, incluindo a análise das necessidades das clínicas odontológicas, a identificação dos requisitos funcionais e não funcionais do sistema e a definição dos critérios de aceitação. O objetivo é garantir que o sistema atenda de forma eficaz e satisfatória às demandas do contexto odontológico, proporcionando uma experiência positiva para os usuários e contribuindo para uma gestão eficiente das clínicas.

### 3.1.1. Identificação das necessidades do usuário

A identificação das necessidades do usuário foi um passo crucial no processo de desenvolvimento do projeto de gerenciamento Odontoclínico. Durante essa etapa, foi reconhecida a importância de um *software* para auxiliar nas atividades rotineiras de uma clínica odontológica, considerando o tempo gasto e a quantidade de tarefas a serem realizadas e distribuídas entre os funcionários. Com base nessa compreensão, foram levantadas as principais funcionalidades necessárias para o sistema, visando otimizar o controle do paciente e da clínica.

O *software* foi projetado para apresentar um painel gerencial (*dashboard*) que oferecesse uma visão geral e no menu, são apresentadas algumas opções personalizadas para diferentes usuários e administradores da clínica. Dentre as funcionalidades identificadas, destacam-se:

- Cadastrar usuários;
- Cadastrar pacientes;
- Cadastrar clínicas e dentistas;
- Acesso aos dados da clínica e dos pacientes;
- Agendamento de consultas (inclusão, edição e remoção);
- Inserção de informações sobre os pacientes (prontuário);

Esses requisitos foram identificados com base na necessidade de otimizar o gerenciamento das atividades de uma clínica odontológica, permitindo um controle mais eficiente do paciente e da clínica, além de facilitar o agendamento de consultas e a geração de documentos relevantes para o tratamento odontológico.

### 3.1.2. Definição de requisitos funcionais e não funcionais

No desenvolvimento de um *software*, a definição dos requisitos funcionais e não funcionais desempenha um papel crucial para garantir o sucesso do projeto. Os requisitos funcionais descrevem as funcionalidades e comportamentos específicos que o sistema deve ter, ou seja, as ações que o sistema deve ser capaz de realizar e as interações que ele deve possibilitar. Já os requisitos não funcionais abrangem aspectos relacionados à qualidade do sistema, como desempenho, segurança, usabilidade e escalabilidade. Ambos os tipos de requisitos são essenciais para garantir que o *software* atenda às necessidades dos usuários, cumpra os objetivos propostos e proporcione uma experiência satisfatória.

Desse modo, durante a etapa de levantamento de requisitos, os requisitos funcionais e não funcionais do *software* foram identificados e organizados em tabelas para uma melhor compreensão e documentação. Na Tabela 1, estão listados os requisitos funcionais do *software*, acompanhados de uma breve descrição, de sua prioridade e dos papéis dos usuários autorizados que irão interagir com cada funcionalidade, usuário ou administrador. Essa tabela fornece uma visão clara das

principais ações e comportamentos esperados do sistema. Já na Tabela 2, são descritos os requisitos não funcionais relevantes para o projeto. A organização dos requisitos em tabelas facilita a visualização e a referência durante o processo de desenvolvimento.

Tabela 1 - Requisitos funcionais

| **Código** | **Nome** | **Descrição** | **Prioridade** | **Papel** |
|---|---|---|---|---|
| RF01 | Cadastro de login para usuário | O funcionário pode realizar um cadastro de usuário para ser possível acessar as funcionalidades do sistema. | Essencial | Administrador |
| RF02 | Edição de login | O funcionário pode editar o cadastro de login de um usuário. | Importante | Administrador |
| RF03 | Deletar login | O funcionário pode deletar o login de um usuário. | Importante | Administrador |
| RF04 | Cadastro de paciente | O funcionário poderá inserir pacientes no sistema. | Essencial | Usuário e Administrador |
| RF05 | Deletar paciente | O funcionário poderá deletar algum paciente cadastrado. | Essencial | Usuário e Administrador |
| RF06 | Editar paciente | Será possível editar informações dos pacientes cadastrados. | Essencial | Usuário e Administrador |
| RF07 | Visualizar paciente | O funcionário poderá visualizar as informações do paciente no sistema. | Importante | Usuário e Administrador |
| RF08 | Cadastro de dentista | O funcionário poderá inserir dentistas no sistema. | Essencial | Administrador |
| RF09 | Deletar dentista | O funcionário poderá deletar algum dentista cadastrado. | Essencial | Administrador |
| RF10 | Editar dentista | Será possível editar informações dos dentistas cadastrados. | Essencial | Administrador |
| RF11 | Visualizar dentista | O funcionário poderá visualizar as informações do dentista no sistema. | Importante | Administrador |
| RF12 | Cadastro de clínica | O funcionário poderá inserir clínicas no sistema. | Essencial | Administrador |
| RF13 | Deletar clínica | O funcionário poderá deletar alguma clínica cadastrada. | Essencial | Administrador |
| RF14 | Editar clínica | Será possível editar informações das clínicas cadastradas. | Essencial | Administrador |
| RF15 | Visualizar clínica | O funcionário poderá visualizar as informações da clínica no sistema. | Importante | Administrador |
| RF16 | Cadastro de prontuário | O dentista pode inserir informações sobre a consulta (relatórios, tratamentos, dentre outros) | Essencial | Administrador |
| RF17 | Deletar prontuário | O funcionário poderá deletar algum prontuário cadastrado. | Essencial | Administrador |
| RF18 | Editar prontuário | Será possível editar informações dos prontuários cadastrados. | Essencial | Administrador |

| | | | | |
|---|---|---|---|---|
| RF19 | Visualizar prontuário | O funcionário poderá visualizar as informações dos prontuários no sistema. | Importante | Administrador |
| RF20 | Agendar consultas | O funcionário poderá agendar consultas no calendário para os pacientes. | Essencial | Usuário e Administrador |
| RF21 | Cancelar consultas | O funcionário poderá cancelar consultas agendadas no calendário. | Essencial | Usuário e Administrador |
| RF22 | Editar consultas | O funcionário poderá fazer modificações nas consultas agendadas. | Importante | Usuário e Administrador |

Tabela 2 – Requisitos não funcionais

| **Código** | **Nome** | **Descrição** | **Categoria** |
|---|---|---|---|
| RNF01 | Segurança | O sistema deve fornecer mecanismos de segurança e autenticação. | Segurança |
| RNF02 | Usabilidade | Usuários deverão operar o sistema após um determinado tempo de treinamento | Usabilidade |
| RNF03 | Portabilidade | O sistema deverá executar em qualquer plataforma e navegador. | Usabilidade |
| RNF04 | Interoperabilidade | O sistema deverá se comunicar com o banco *SQL* | Interoperabilidade |
| RNF05 | Controle de acesso | Apenas os usuários cadastrados terão acesso ao sistema | Segurança |
| RNF06 | Permissões | Apenas os usuários com as devidas permissões acessarão funcionalidades restritas. | Segurança |

### 3.1.3. Identificação dos atores

A identificação dos atores em um sistema é de suma importância para o desenvolvimento e funcionamento adequado do sistema. Os atores representam os diferentes usuários ou entidades que interagem com o sistema e desempenham papéis específicos, compreender os diferentes atores e suas necessidades é fundamental para o *design* e implementação de um sistema que atenda às demandas e expectativas de cada usuário. Além disso, a identificação dos atores auxilia na definição de permissões e restrições de acesso, garantindo a segurança e integridade dos dados. Ao reconhecer os atores envolvidos, é possível desenvolver

um sistema mais eficiente, intuitivo e adaptado às necessidades de cada usuário, contribuindo para uma experiência positiva e produtiva.

O principal ator no sistema em questão é o Administrador, que possui acesso completo a todas as funcionalidades do sistema. O Administrador é responsável por cadastrar pacientes, dentistas e clínicas, cadastrar prontuários dos pacientes, agendar, editar e cancelar consultas no calendário e realizar a gestão dos usuários do sistema. Além do Administrador, existem outros usuários com acesso restrito. Esses usuários têm permissões limitadas e podem editar suas próprias informações de *login*, cadastrar, editar, excluir e visualizar pacientes, agendar, editar e cancelar consultas no calendário.

Os usuários com perfil de Administrador são geralmente donos, gerentes e dentistas da clínica, enquanto os outros usuários são funcionários e secretárias que desempenham funções específicas na clínica odontológica. A identificação clara dos atores e seus respectivos papéis é essencial para o correto funcionamento e segurança do sistema, garantindo que cada usuário tenha acesso apenas às funcionalidades necessárias para desempenhar suas atividades.

O pacote "*Laravel-Permission*" desenvolvido pela biblioteca *Spatie* é uma ferramenta amplamente utilizada para implementar controle de acesso e gerenciamento de permissões em projetos *Laravel*. Com o auxílio desse pacote, foi possível atribuir permissões aos diferentes usuários do sistema, permitindo um controle sobre as ações e recursos disponíveis para cada um. Através do uso do pacote, é possível criar roles (papéis) e permissões personalizadas, atribuí-las aos usuários e realizar verificações de permissões em tempo de execução.

### 3.1.4. Definição dos casos de uso

Os casos de uso desempenham um papel fundamental no processo de desenvolvimento de um projeto, pois capturam as interações entre os atores (usuários) e o sistema, eles descrevem os principais cenários de uso, especificando as ações que os usuários podem realizar e as respostas esperadas do sistema. Através dos casos de uso, é possível obter uma visão clara e abrangente dos

requisitos funcionais do sistema, identificando os fluxos de trabalho e as funcionalidades necessárias para atender às necessidades dos usuários.

No contexto deste projeto, os diagramas de casos de uso foram elaborados utilizando a ferramenta *Lucidchart*, uma plataforma de diagramação baseada na *web*, essa escolha foi motivada pela facilidade de uso e pelos recursos que a ferramenta oferece para representar visualmente as interações entre os atores e o sistema. Os diagramas de casos de uso mapeiam as principais funcionalidades da aplicação, destacando as ações disponíveis para os usuários e as respostas correspondentes do sistema. É importante ressaltar que o acesso e a execução das ações descritas nos casos de uso exigem que os usuários realizem o *login* no sistema.

Para uma visualização completa dos diagramas de casos de uso, eles estão disponíveis no Apêndice A deste trabalho, proporcionando uma compreensão abrangente das interações entre os atores e o sistema, contribuindo para o desenvolvimento de um *software* eficiente e alinhado às necessidades dos usuários.

## 3.2. Design de interface

No design de *interface*, foram adotadas diversas estratégias para garantir uma experiência de usuário intuitiva e agradável. A tela de *login* foi projetada com um *layout* minimalista e limpo, com o foco direcionado para o campo de *login*, centralizado na tela, e acompanhado da logo da Odonto *Sistem* para identificação visual. Após o *login*, a tela principal foi concebida para proporcionar uma visão geral das funcionalidades do sistema.

Para facilitar o acesso às configurações individuais e permitir uma navegação fluida, foi incluído um ícone de usuário no canto superior direito, possibilitando ao usuário acessar seu perfil e sair do sistema com facilidade. Na lateral esquerda da tela principal, abaixo da logo, uma barra de menu foi inserida, contendo todas as funcionalidades do sistema. Os menus estão organizados de forma lógica e intuitiva, incluindo opções como pacientes, prontuários, agenda, gerenciamento de clínicas e dentistas, além das configurações, que englobam o perfil de usuário e a gestão de usuários.

No que diz respeito às cores, a escolha das cores desempenha um papel importante na criação de uma identidade visual adequada e na transmissão de mensagens específicas. O uso de tons de rosa como cor principal no *design* do sistema pode ser uma estratégia interessante.

O rosa é uma cor frequentemente associada a características como delicadeza, feminilidade, tranquilidade e calma. Essas qualidades podem ser especialmente relevantes em um ambiente odontológico, onde muitos pacientes podem sentir ansiedade ou desconforto. Ao utilizar tons de rosa no *design* da interface, busca-se transmitir uma sensação de acolhimento, suavidade e serenidade, contribuindo para criar um ambiente mais amigável e confortável para os pacientes.

A logo da *Odonto Sistem* foi especialmente desenvolvida, tendo como imagem principal um dente, representando de forma icônica o contexto de uma clínica odontológica. O *design* de interface cuidadosamente planejado busca proporcionar uma experiência visualmente agradável, ao mesmo tempo em que facilita a navegação e o acesso às funcionalidades do sistema. Essas escolhas foram feitas levando em consideração a usabilidade, a identidade visual e a intuitividade, buscando atender às necessidades dos usuários e proporcionar uma experiência satisfatória ao utilizar o *software*.

## 3.3. Desenvolvimento de funcionalidades

No contexto do desenvolvimento do sistema, é fundamental descrever em detalhes as funcionalidades específicas que foram implementadas. Neste tópico, serão abordadas as diversas funcionalidades do sistema, incluindo a implementação de recursos e a integração dos componentes necessários para garantir o pleno funcionamento do sistema. Serão apresentados os principais módulos e suas respectivas funcionalidades, bem como a modelagem do banco de dados. A partir dessas informações, será possível compreender como o sistema foi projetado e construído, destacando os aspectos técnicos e as soluções adotadas para atender às necessidades dos usuários.

### 3.3.1. Modelagem do banco de dados

A modelagem do banco de dados desempenha um papel fundamental no desenvolvimento de um sistema, pois permite a organização e estruturação das informações que serão armazenadas. Para essa finalidade, utiliza-se o Diagrama Entidade-Relacionamento (*DER*), uma ferramenta amplamente utilizada na modelagem de bancos de dados, incluindo a aplicação do *Laravel*.

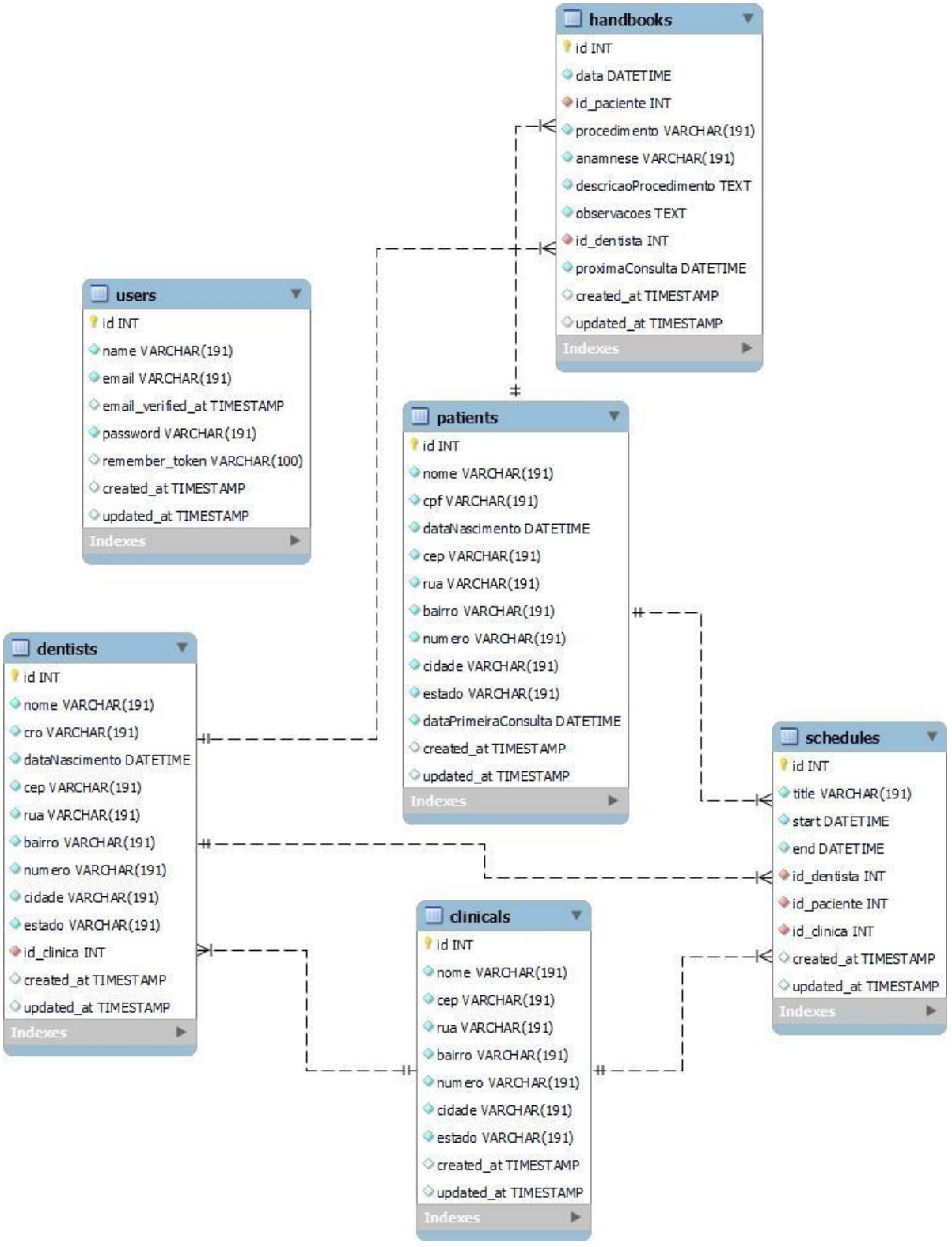
O *DER* é um modelo visual que representa as entidades (tabelas), seus atributos e os relacionamentos entre elas. Ele fornece uma visão clara e intuitiva da estrutura do banco de dados, permitindo identificar as entidades, seus atributos e as relações entre elas. O *DER* é composto por entidades, que representam as tabelas do banco de dados, e pelos relacionamentos, que indicam como as entidades estão conectadas.

Ao utilizar o *MySQL Workbench*, uma ferramenta de modelagem de bancos de dados, foi possível criar o *DER* da aplicação, o qual ajudou a definir a estrutura das tabelas, seus atributos e os relacionamentos entre elas, proporcionando uma base sólida para a implementação do banco de dados. Além disso, o *DER* facilita a compreensão e a manutenção do banco de dados ao longo do desenvolvimento do projeto, fornecendo uma representação visual clara e organizada das informações.

No contexto do *Laravel*, é importante mencionar que o *framework* adota algumas convenções, como o uso de nomes de tabelas no plural e a utilização de um atributo "*id*" como chave primária padrão. Essas convenções auxiliam na padronização e no entendimento do modelo de dados, tornando-o mais legível e consistente. O DER completo está disponível no Apêndice B.

A Figura 1 representa as funcionalidades de usuários, dentistas, pacientes, clínicas, prontuários e agenda.

Figura 1 - DER – Usuários, Dentistas, Pacientes, Clínicas, Prontuários e Agenda




Elaborado pela autora

A modelagem do banco de dados contempla diversas tabelas que armazenam informações relevantes para o sistema. A tabela “usuários” contém os dados principais de um usuário, como nome, email, verificação de email, senha e verificação de senha.

A tabela “*dentists*” possui informações específicas dos profissionais de odontologia, como nome, número de registro no Conselho Regional de Odontologia (*CRO*), data de nascimento e endereço completo com CEP, rua, bairro, número, cidade e estado. Além disso, a tabela possui a chave estrangeira “id_clinica”, que indica a qual clínica o dentista está associado.

A tabela “*clinicals*” contém informações referentes à clínica odontológica, como nome, endereço completo com CEP, rua, bairro, número, cidade e estado.

A tabela “*patients*” armazena os dados dos pacientes, como nome, CPF, data de nascimento e endereço completo com CEP, rua, bairro, número, cidade e estado. Essa tabela também registra a data da primeira consulta do paciente.

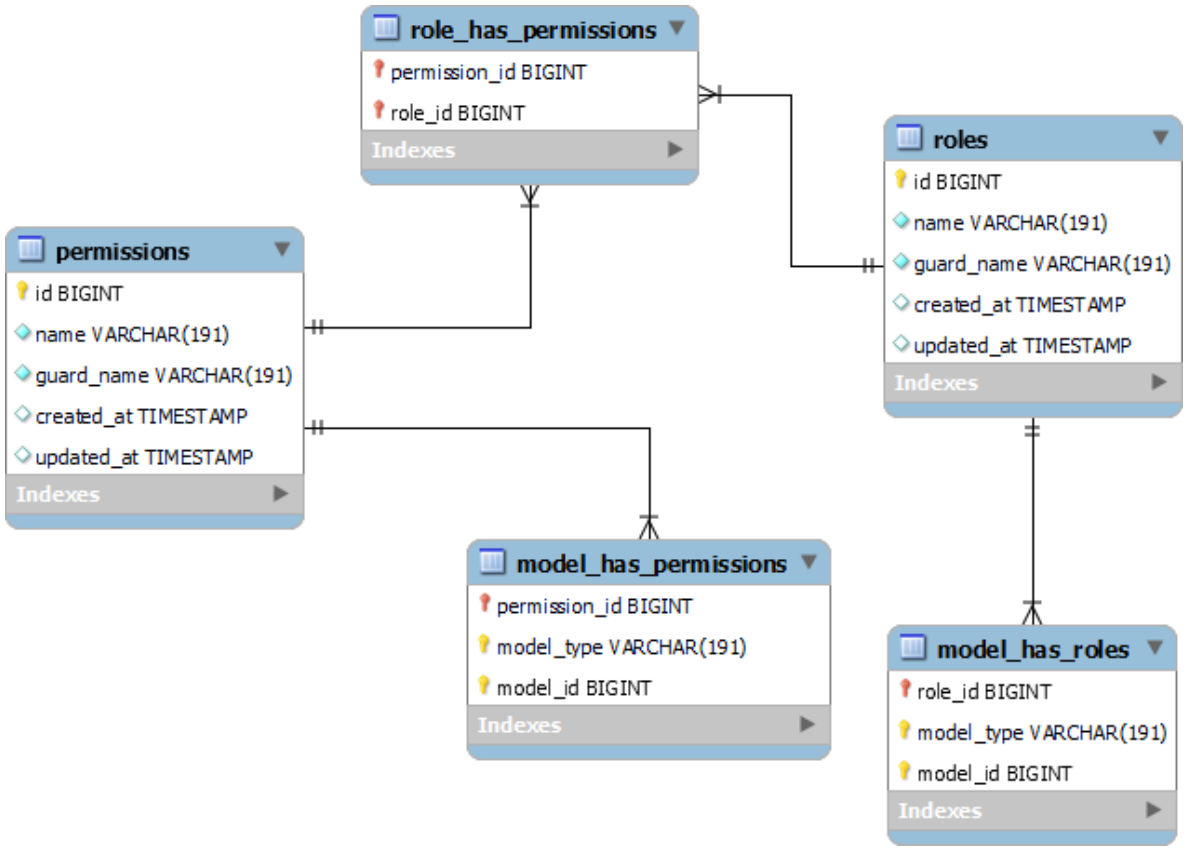
A tabela “*handbooks*” registra as informações dos prontuários, incluindo as chaves estrangeiras “id_paciente” e “id_dentista”, que fazem referência às tabelas de paciente e dentista, respectivamente. Além disso, essa tabela armazena a data do procedimento realizado, a anamnese, a descrição do procedimento, observações relevantes e a data da próxima consulta.

Por fim, a tabela “*schedules*” registra os agendamentos de consultas, contendo o título (nome do paciente), a data e hora de início e fim da consulta. Essa tabela também possui as chaves estrangeiras “id_paciente”, “id_clinica” e “id_dentista”, que indicam as referências às tabelas correspondentes.

Essas tabelas foram projetadas de forma a armazenar os dados necessários para o funcionamento do sistema de gestão de clínicas odontológicas, proporcionando um armazenamento organizado e estruturado das informações relacionadas aos usuários, dentistas, clínicas, pacientes, prontuários e agendamentos de consultas.

No contexto do sistema, é necessário atribuir diferentes papéis e permissões aos usuários para controlar o acesso e as funcionalidades disponíveis. Essa gerência de papéis e permissões é realizada por meio das tabelas de “funções” (*roles*) e “permissões” (*permissions*), que são criadas e gerenciadas pelo pacote *Laravel-permission*, como demonstrado na Figura 2.

Figura 2 - DER – Roles, Permissions




Elaborado pela autora

### 3.3.2. Tela principal

A tela inicial da aplicação é projetada para fornecer informações gerais relevantes sobre o sistema e seus dados. Um dos principais elementos dessa tela é um gráfico que apresenta a quantidade de atendimentos realizados ao longo do tempo, sendo possível visualizar essa informação por mês ou de forma anual. Ao posicionar o *mouse* sobre um dos meses no gráfico, é exibido o número específico

de atendimentos correspondente a esse período. Essa visualização gráfica permite uma rápida compreensão e análise dos dados de atendimento, oferecendo uma visão geral da produtividade da clínica ao longo do tempo.

A Figura 3 exibe um trecho do código referente à implementação da tela do *Dashboard* no projeto.

Figura 3 - Trecho de código da tela do Dashboard – Desempenho Geral

```
<div class="row">
    <div class="col-12">
        <div class="card card-chart">
            <div class="card-header ">
                <div class="row">
                    <div class="col-sm-6 text-left">
                        <h5 class="card-category">Total de Atendimentos Odontológicos</h5>
                        <h2 class="card-title">Desempenho Geral</h2>
                    </div>
                    <div class="col-sm-6">
                        <div class="btn-group btn-group-toggle float-right" data-toggle="buttons">
                            <label class="btn btn-sm btn-primary btn-simple active" id="0">
                                <input type="radio" name="options" checked>
                                <span class="d-none d-sm-block d-md-block d-lg-block d-xl-block">Mensal</span>
                                <span class="d-block d-sm-none">
                                    <i class="tim-icons icon-single-02"></i>
                                </span>
                            </label>
                            <label class="btn btn-sm btn-primary btn-simple" id="1">
                                <input type="radio" class="d-none d-sm-none" name="options">
                                <span class="d-none d-sm-block d-md-block d-lg-block d-xl-block">Anual</span>
                                <span class="d-block d-sm-none">
                                    <i class="tim-icons icon-gift-2"></i>
                                </span>
                            </label>
                        </div>
                    </div>
                </div>
            </div>
            <div class="card-body">
                <div class="chart-area">
                    <canvas id="chartBig1"></canvas>
                </div>
            </div>
        </div>
    </div>
</div>
```

Elaborado pela autora

O trecho de código apresenta a estrutura básica de um *card* que exibe um gráfico, nele há um título e um subtítulo que fornecem informações sobre o gráfico. Na parte superior direita do *card*, são exibidos dois botões que permitem alternar entre as representações mensal e anual do gráfico.

O elemento “<*canvas*>“ serve como o contêiner onde o gráfico é exibido na interface do usuário e é utilizado para renderizar o gráfico propriamente dito, o gráfico em si é desenhado utilizando tecnologias como *JavaScript* e bibliotecas de visualização de dados. Essa estrutura do *card* com o título, subtítulo, botões e “<*canvas*>“ permite apresentar de forma organizada e visualmente agradável o gráfico correspondente, facilitando a compreensão dos dados pelos usuários do sistema.

A Figura 4 e Figura 5 apresentam trechos de código *JavaScript* responsáveis pela construção do gráfico de desempenho geral, com foco inicial na visualização mensal.

Figura 4 - Trecho de código arquivo JavaScript – Desempenho Geral

```
// Fazer uma requisição AJAX para obter os dados do banco de dados
$.ajax({
    url: "/consultas-por-mes", // Defina a URL correta para a rota do Laravel que retorna os dados
    method: "GET",
    success: function (response) {
        // Os dados foram recebidos com sucesso
        var chart_labels = [
            "JAN",
            "FEB",
            "MAR",
            "ABR",
            "MAI",
            "JUN",
            "JUL",
            "AGO",
            "SET",
            "OUT",
            "NOV",
            "DEZ",
        ];
        var chart_data = response.data;

        var ctx = document.getElementById("chartBig1").getContext("2d");

        var gradientStroke = ctx.createLinearGradient(0, 230, 0, 50);
        gradientStroke.addColorStop(1, "rgba(72,72,176,0.1)");
        gradientStroke.addColorStop(0.4, "rgba(72,72,176,0.0)");
        gradientStroke.addColorStop(0, "rgba(119,52,169,0)"); //purple colors
```

Elaborado pela autora

Figura 5 - Trecho de código arquivo JavaScript – Desempenho Geral

```
var config = {
    type: "line",
    data: {
        labels: chart_labels,
        datasets: [
            {
                label: "Atendimentos",
                fill: true,
                backgroundColor: gradientStroke,
                borderColor: "#d346b1",
                borderWidth: 2,
                borderDash: [],
                borderDashOffset: 0.0,
                pointBackgroundColor: "#d346b1",
                pointBorderColor: "rgba(255,255,255,0)",
                pointHoverBackgroundColor: "#d346b1",
                pointBorderWidth: 20,
                pointHoverRadius: 4,
                pointHoverBorderWidth: 15,
                pointRadius: 4,
                data: chart_data,
            },
        ],
    },
    options: gradientChartOptionsConfigurationWithTooltipPurple,
};

// Renderizar o gráfico com os dados carregados do banco de dados
var myChartData = new Chart(ctx, config);

$("#0").click(function () {
    var data = myChartData.config.data;
    data.datasets[0].data = chart_data;
    data.labels = chart_labels;
    myChartData.update();
});
```

Elaborado pela autora

O código apresentado é uma chamada *AJAX* para recuperar os dados dos atendimentos por mês do servidor, ele utiliza o método “$.*ajax*()” do *jQuery* para enviar uma solicitação *HTTP GET* para a rota /consultas-por-mes no servidor.

Uma vez que a solicitação é bem-sucedida, a função “*success*” é executada. Nessa função, o código manipula os dados recebidos e configura as opções do gráfico.

- O *array* “*chart_labels*” contém os rótulos dos meses do ano.
- A variável “*chart_data*” contém os dados dos atendimentos por mês, obtidos da resposta da requisição *AJAX*.

Em seguida, o código utiliza o “*document*.*getElementById*()” para obter o elemento “<*canvas*>“ do gráfico, com o ID “*chartBig1*”. A variável “*ctx*” representa o contexto do elemento do “<*canvas*>“ e é usada para criar um gradiente de cores para o preenchimento do gráfico. O objeto “*config*” define as configurações do gráfico, incluindo os rótulos, dados, cores e estilo. Ele utiliza o tipo de gráfico “*line*” (linha) e define um único conjunto de dados chamado “Atendimentos”.

Por fim, o código cria uma instância do gráfico utilizando a *classe Chart* do *Chart*.js, passando o contexto e a configuração definida anteriormente, ele também define um manipulador de eventos para o elemento com o *ID* 0 (zero), que atualiza o gráfico quando clicado. Em resumo, esse trecho de código realiza uma chamada *AJAX* para obter os dados dos atendimentos por mês do servidor e utiliza o *Chart*.js para renderizar um gráfico de linha interativo no elemento “<*canvas*>“.

A Figura 6 representa um trecho de código em *JavaScript* responsável por modificar a visualização do gráfico de atendimentos para uma exibição anual. O código é acionado quando o elemento com o ID 1 (um) é clicado.

Figura 6 - Trecho de código arquivo JavaScript – Desempenho Geral

```
        $("#1").click(function () {
            $.ajax({
                url: "/consultas-por-ano",
                method: "GET",
                async: false,
                success: function (response) {
                    var currentYear = new Date().getFullYear();
                    var chart_labels = [];

                    for (
                        var i = currentYear - 4;
                        i <= currentYear;
                        i++
                    ) {
                        chart_labels.push(i);
                    }

                    var chart_data = response.data;

                    // Atualiza o gráfico com os novos dados
                    var data = myChartData.config.data;
                    data.datasets[0].data = chart_data;
                    data.labels = chart_labels;
                    myChartData.update();
                },
                error: function (xhr, status, error) {
                    console.log(error);
                },
            });
        });
    },
    error: function (xhr, status, error) {
        // Ocorreu um erro ao obter os dados do banco de dados

        console.log(
            "Erro ao obter os dados do banco de dados: " + error
        );
    },
});
```

Elaborado pela autora

O trecho de código apresentado representa um manipulador de eventos em *JavaScript* que é acionado quando o elemento com o ID “1” é clicado. Dentro dessa

função, é realizado uma requisição *AJAX* assíncrona para uma determinada *URL* (“/consultas-por-ano”) utilizando o método *GET*.

Quando a requisição é concluída com sucesso, a função de sucesso (*success*) é executada. Nessa função, é recuperado o ano atual usando o objeto *Date*(), e em seguida, um *loop* é executado para preencher o *array* `*chart_labels*` com os anos dos últimos cinco anos, incluindo o ano atual.

Os dados do gráfico são obtidos a partir da resposta da requisição *AJAX* e armazenados na variável `*chart_data*`. Em seguida, os dados e os rótulos do gráfico são atualizados com as novas informações. A variável `data` é atribuída com as informações do gráfico atual (`*myChartData*.*config*.*data*`), os dados são atualizados para os valores de `*chart_data*`, e os rótulos são atualizados para os valores de `*chart_labels*`. A função `*myChartData*.*update*()` é chamada para atualizar o gráfico com os novos dados e rótulos.

Em caso de erro na requisição *AJAX*, a função de erro (*error*) é executada e o erro é exibido no console. Esse trecho de código permite a atualização dos dados e rótulos do gráfico de atendimentos para exibir os dados por ano, proporcionando uma funcionalidade dinâmica ao usuário.

É fundamental enfatizar a elaboração da consulta ao banco de dados que recupera os dados necessários para o gráfico em questão. Com isso, temos o c*ontroller* da tela, cuja responsabilidade é recuperar os dados do banco de dados e enviá-los para o *JavaScript*. Na função “consultaPorMês” verifica-se o ano atual ou recebe um ano específico fornecido na requisição *HTTP*, em seguida, realiza uma consulta ao banco de dados utilizando o modelo “*Schedule*”. Essa consulta seleciona o mês a partir do campo “*start*” e conta o número total de consultas por mês.

Para filtrar os resultados apenas para o ano especificado, a função utiliza o método “*whereYear*” do *Eloquent*. Em seguida, agrupa os resultados por mês utilizando o método “*groupBy*” e os ordena em ordem crescente de mês com o método “*orderBy*”. Uma vez que a consulta é executada no banco de dados, os

resultados são retornados como uma coleção de objetos. Em seguida, os dados são formatados para serem retornados como uma resposta *JSON*.

A função “*consultasPorAno*” é responsável por buscar os dados no banco de dados e retornar a contagem de consultas por ano, em um intervalo específico de anos. Primeiro, a função obtém o ano atual utilizando a função “*date*('Y')” e em seguida, define o número de anos que serão retornados, por meio da variável “$*numYears*”. É calculado o limite inferior do intervalo de anos, utilizando o valor do ano atual e subtraindo o número de anos desejado, acrescentando 1 (um).

Posteriormente, é feita uma consulta ao banco de dados usando o modelo “*Schedule*”, a consulta seleciona o ano (extraído do campo “*start*”) e conta o número total de consultas por ano. A cláusula “*whereYear*” é utilizada para filtrar os resultados, considerando apenas os anos maiores ou iguais ao limite inferior definido anteriormente, os resultados da consulta são agrupados por ano, utilizando o método “*groupBy*”, e ordenados em ordem crescente de ano, com o método “*orderBy*”. Após executar a consulta no banco de dados, os resultados são retornados como um *JSON*.

Na tela principal da aplicação, além do gráfico de desempenho dos atendimentos, também é exibido um segundo gráfico que apresenta informações gerais sobre a quantidade de pacientes nas clínicas odontológicas. Esse gráfico é dividido por estados, permitindo visualizar a distribuição dos pacientes em cada região, ao posicionar o cursor do *mouse* sobre o gráfico, é possível obter informações específicas sobre a quantidade de pacientes em cada estado. Além disso, acima do gráfico, é exibido o número total de pacientes, proporcionando uma visão geral da quantidade de pacientes cadastrados no sistema.

Essa representação visual dos dados é valiosa para que os usuários possam ter uma compreensão rápida e intuitiva da situação dos pacientes nas clínicas e identificar possíveis tendências ou variações nos números. A Figura 7 e Figura 8 representa um trecho de código *JavaScript* responsável por construir o gráfico apresentado no “*card*” mencionado anteriormente.

Figura 7 - Trecho de código arquivo JavaScript - Pacientes por estado

```
$.ajax({
    url: "/pacientes-por-estado",
    method: "GET",
    success: function (response) {
        var chart_labels = response.data.siglas;
        var chart_data = response.data.quantidades;

        var ctx = document
            .getElementById("CountryChart")
            .getContext("2d");

        var gradientStroke = ctx.createLinearGradient(0, 230, 0, 50);

        gradientStroke.addColorStop(1, "rgba(29,140,248,0.2)");
        gradientStroke.addColorStop(0.4, "rgba(29,140,248,0.0)");
        gradientStroke.addColorStop(0, "rgba(29,140,248,0)"); //blue colors
```

Elaborado pela autora

Figura 8 - Trecho de código arquivo JavaScript - Pacientes por estado

```
        var myChart = new Chart(ctx, {
            type: "bar",
            responsive: true,
            legend: {
                display: false,
            },
            data: {
                labels: chart_labels,
                datasets: [
                    {
                        label: "Pacientes",
                        fill: true,
                        backgroundColor: gradientStroke,
                        hoverBackgroundColor: gradientStroke,
                        borderColor: "#1f8ef1",
                        borderWidth: 2,
                        borderDash: [],
                        borderDashOffset: 0.0,
                        data: chart_data,
                    },
                ],
            },
            options: gradientBarChartConfiguration,
        });
    },
    error: function (xhr, status, error) {
        console.log(
            "Erro ao obter os dados do banco de dados: " + error
        );
    },
});
```

Elaborado pela autora

No trecho de código apresentado, é realizado uma requisição *AJAX* para a rota “/pacientes-por-estado” utilizando o método *GET*, o objetivo dessa requisição é obter os dados necessários para a construção do gráfico de pacientes por estado.

Quando a requisição é bem-sucedida, a função “*success*” é acionada, e a variável “*response*” contém os dados retornados pelo servidor, nesse caso, os dados são separados em duas variáveis: “*chart_labels*” que armazena as siglas dos estados e “*chart_data*” que armazena as quantidades de pacientes por estado.

Em seguida, é obtido o contexto do elemento “<*canvas*>“ com o ID “*CountryChart*” utilizando “*document*.*getElementById*()”, esse contexto é necessário para criar a instância do gráfico, utilizando “*new Chart*(*ctx*, *options*)”. O tipo de gráfico é definido como “*bar*” (gráfico de barras) e são especificadas as configurações relacionadas aos dados, como rótulos, cores e valores das barras.

Por fim, o gráfico é renderizado no elemento “<*canvas*>“ e as opções de configuração são aplicadas, em caso de erro na requisição, a função “*error*” é acionada, exibindo uma mensagem de erro no console. Esse trecho de código é responsável por buscar os dados sobre pacientes por estado e criar o gráfico de barras correspondente para visualização na aplicação.

Compreender a função responsável por recuperar os dados e enviá-los para o arquivo *JavaScript* é de grande relevância. A função “*pacientesPorEstado*” desempenha um papel importante ao recuperar os dados relacionados à quantidade de pacientes por estado e disponibilizá-los para uso no *front-end*. Ao executar a função, ocorre uma consulta no modelo “*Patients*” para selecionar o campo “estado” e a contagem total de pacientes por estado, utilizando uma expressão *SQL* personalizada, os resultados são agrupados por estado e obtidos como uma coleção de objetos.

Em seguida, os valores dos campos “estado” e “*total_pacientes*” são extraídos dos resultados e convertidos em *arrays* utilizando o método “*pluck*”, esses *arrays* são utilizados para preencher um novo *array* chamado “$dados”, que possui duas chaves: “quantidades” e “siglas”. A chave “quantidades” armazena o *array* com as quantidades de pacientes por estado, enquanto a chave “siglas” armazena o *array*

com as siglas dos estados. Por fim, a função retorna uma resposta *HTTP* em formato *JSON*, onde os dados são encapsulados dentro da chave 'data'.

Concluindo, é exibida uma tabela contendo os agendamentos de pacientes do dia, apresentando informações como o nome do paciente, data e horário do agendamento. Essa tabela permite que os usuários visualizem de forma organizada e atualizada os compromissos agendados para o dia em questão, além disso, na lateral superior direita da tela, há um botão que oferece a opção de redirecionar o usuário para a tela de agenda, permitindo um acesso rápido e fácil para gerenciar os agendamentos. Ao lado do botão, é evidenciado o dia atual, proporcionando uma referência visual para os usuários.

As Figuras 9 e 10 ilustram o trecho de código responsável pela criação e preenchimento da tabela, garantindo a exibição correta dos agendamentos na interface do sistema.

Figura 9 - Trecho de código da tela do Dashboard – Pacientes por estado

```
<div class="col-lg-6 col-md-12">
    <div class="card card-chart h-100">
        <div class="card-header">
            <div class="row">
                <div class="col-sm-6 text-left">
                    <h5 class="card-category">Consultas do dia</h5>

                    @php
                        $now = \Carbon\Carbon::now('America/Sao_Paulo');
                        $formattedDate = $now->format('d/m/Y');
                    @endphp
                    <h3 class="card-title"><i class="tim-icons icon-calendar-60 text-info"></i> {{ $formattedDate }}
                    </h3>
                </div>
                <div class="col-sm-6">
                    <div class="btn-group btn-group-toggle float-right" data-toggle="buttons">
                        <label class="btn btn-sm btn-primary btn-simple active" id="botãoAgenda">
                            <input type="radio" name="options" checked>
                            <span class="d-none d-sm-block d-md-block d-lg-block d-xl-block">Agenda</span>
                            <span class="d-block d-sm-none">
                                <i class="tim-icons icon-single-02"></i>
                            </span>
                        </label>
                    </div>
                </div>
            </div>
        </div>
```

Elaborado pela autora

Na parte superior esquerda do *card*, é exibido o título “Consultas do dia”, logo abaixo, é gerado dinamicamente, por meio do código em *PHP*, a data atual no formato “*dd*/*mm*/*aaaa*”, essa data representa o dia em que as consultas estão sendo

exibidas. Na parte superior direita do *card*, há um botão que possui a funcionalidade de redirecionar o usuário para a tela de agenda, esse botão está estilizado como um grupo de botões de alternância, com um botão ativo por padrão.

Figura 10 - Trecho de código da tela do Dashboard – Pacientes por estado

```
<div class="card-body">
    <div class="table-responsive h-100">
        <table class="table tablesorter" id="">
            <thead class=" text-primary">
                <tr>
                    <th>
                        Nome
                    </th>
                    <th>
                        Data
                    </th>
                    <th>
                        Horário
                    </th>
                </tr>
            </thead>
            <tbody>
                @foreach ($agendamentos as $agendamento)
                    <tr>
                        <td>
                            {{ $agendamento->title }}
                        </td>
                        <td>
                            {{ date('d/m/Y', strtotime($agendamento->start)) }}
                        </td>
                        <td>
                            {{ date('H:i', strtotime($agendamento->start)) }} -
                            {{ date('H:i', strtotime($agendamento->end)) }}
                        </td>
                    </tr>
                @endforeach
            </tbody>
        </table>
    </div>
</div>
```

Elaborado pela autora

Através de um laço de repetição em *PHP*, os dados dos agendamentos são percorridos e exibidos em linhas da tabela. Para cada agendamento, são exibidos o nome do paciente, a data no formato “*dd*/*mm*/*aaaa*” e o horário no formato “*hh*:*mm* - *hh*:*mm*”, indicando o início e o fim da consulta. É importante destacar que os dados exibidos na tabela são trazidos para o arquivo *Blade PHP* a partir do Controller da tela, o *Controller* é responsável por realizar consultas no banco de dados e obter os agendamentos de pacientes do dia, assim como demonstrado na Figura 11.

Figura 11 - Home Controller

```php
/**
 * Show the application dashboard.
 *
 * @return \Illuminate\View\View
 */
public function index()
{
    config(['app.name' => 'Odonto Sistem | Home']);

    $currentDate = Carbon::now('America/Sao_Paulo')->toDateString();
    $agendamentos = $this->objSchedule
        ->whereDate('start', $currentDate)
        ->orderBy('start')
        ->get();

    $totalPacientes = Patients::count();
    return view('dashboard')->with(['total_pacientes' => $totalPacientes])->with(['agendamentos' => $agendamentos]);
}
```

Elaborado pela autora

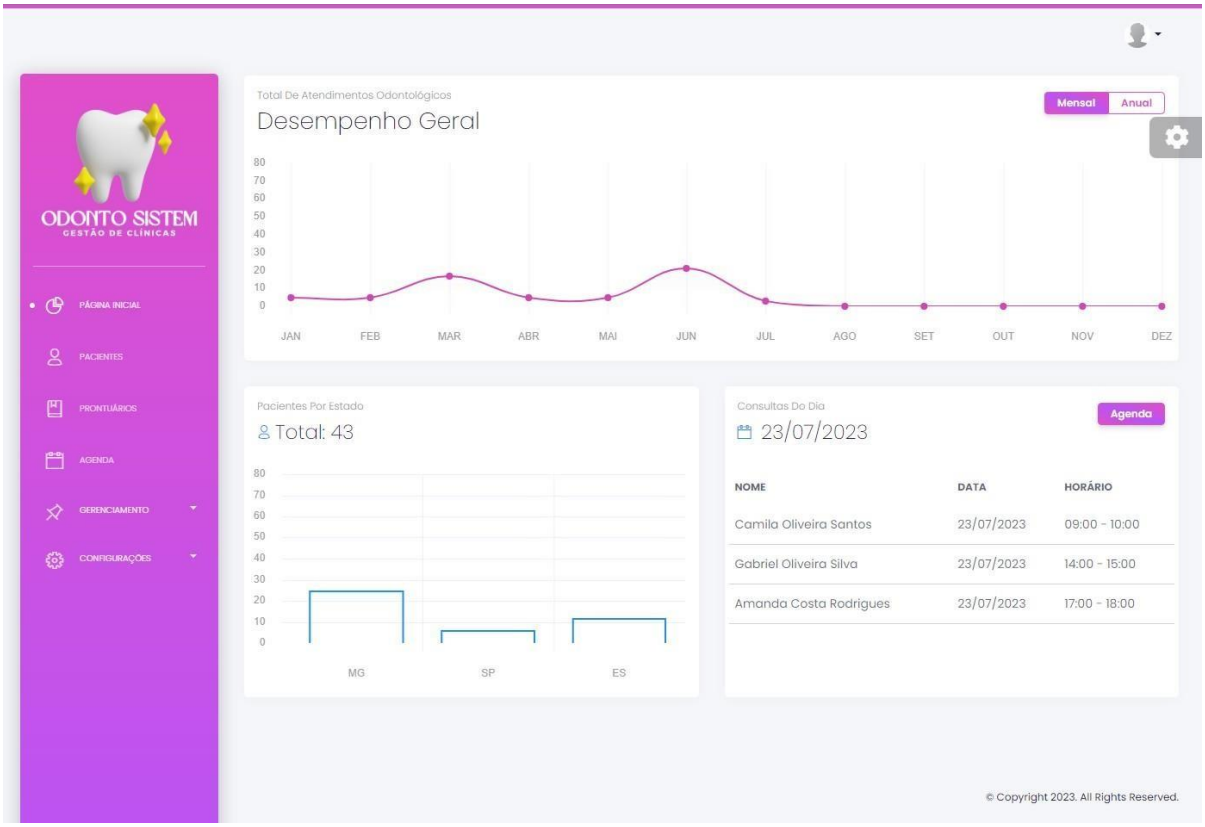
No trecho de código apresentado, a função “*index*()” é responsável por retornar as informações necessárias para a tela inicial. Primeiramente, a variável “$*currentDate*” é utilizada para armazenar a data atual no fuso horário “*America*/*Sao_Paulo*” no formato de data “*aaaa*/*mm*/*dd* “. Essa data é utilizada para filtrar os agendamentos do dia na próxima etapa.

A variável “$agendamentos” é preenchida através da chamada ao método “*whereDate*()” da instância da classe *Schedule* atribuída na propriedade “$*objSchedule*”, que representa a tabela responsável pelos agendamentos. Essa chamada realiza uma consulta no banco de dados para obter todos os agendamentos com a data de início igual à data atual. Os agendamentos são ordenados por ordem de início através do método “*orderBy*(*'start'*)” e, em seguida, são obtidos através do método “*get*()”. A variável “$*totalPacientes*” recebe o resultado da contagem de registros na tabela de pacientes utilizando o modelo “*Patients*”, utilizando esse valor no gráfico anterior.

Por fim, a função retorna a *view* “*dashboard*” com as informações dos total de pacientes e dos agendamentos do dia através dos métodos “*with*()”. Dessa forma, ao acessar a página inicial, essas informações serão disponibilizadas para serem utilizadas na construção do gráfico e da tabela.

A Figura 12 retrata a tela de dashboard, onde se encontram os gráficos que apresentam uma visão geral dos atendimentos e pacientes, bem como a tabela de atendimentos diários, dispostos de forma organizada e cronológica.

Figura 12 - Tela Dashboard




Elaborado pela autora

### 3.3.3. Telas de cadastro

O *CRUD* é um acrônimo para as quatro operações básicas de um sistema: Criar (*Create*), Ler (*Read*), Atualizar (*Update*) e Deletar (*Delete*), ao desenvolver as telas de cadastro do sistema com base no *CRUD*, foi possível implementar uma abordagem completa e padrão para a gestão de dados. As telas de paciente, dentista, clínica e prontuários seguem essas operações, permitindo a criação de novos registros, a leitura e visualização dos dados existentes, a atualização de informações e a exclusão de registros, quando necessário. Essa estrutura oferece uma base sólida para interação com o banco de dados e facilita a administração e

organização dos dados em todas as áreas do sistema, proporcionando uma experiência mais consistente e eficiente aos usuários.

#### 3.3.3.1. Tela inicial de pacientes

A tela de pacientes é um excelente exemplo para ilustrar a estrutura das telas de cadastro. Seguindo o padrão *CRUD*, foi desenvolvida uma página completa que possibilita a criação de novos pacientes, permitindo a inserção de informações relevantes, como nome, data de nascimento e detalhes de endereço, no sistema. Além disso, a tela oferece a funcionalidade de visualizar o cadastro de um paciente, bem como a opção de editar ou excluir o registro, caso seja necessário. A Figura 13 representa a criação dessa estrutura no arquivo "*blade.php*", onde os elementos e recursos são organizados para proporcionar uma experiência intuitiva e eficiente ao usuário.

Figura 13 – Cabeçalho - Tela Paciente

```
<div class="card-header">
    <div class="row">
        <div class="col-sm-8 text-left">
            <h2 class="card-title">Pacientes</h2>
        </div>
        {{-- Buttons --}}
        <div class="col-sm-4">
            <div class="btn-group btn-group-toggle float-right">
                <div class="text-right">
                    <a href="{{ url('patients/create') }}" class="btn btn-sm btn-primary">Cadastrar</a>
                </div>
            </div>
        </div>
    </div>
</div>
```

Elaborado pela autora

Esse trecho de código cria um cabeçalho de seção para a página de pacientes, exibindo o título "*Pacientes*" e um botão de "Cadastrar" à direita para adicionar novos pacientes. A utilização de classes do *Bootstrap* torna a exibição mais organizada e adaptável a diferentes tamanhos de tela.

A Figura 14 cria a barra de busca na página de pacientes, permitindo que os usuários encontrem rapidamente um paciente específico em meio a uma grande quantidade de registros cadastrados. Essa funcionalidade melhora a usabilidade e a produtividade do sistema.

Figura 14 – Busca - Tela paciente

```
<div class="card-header">
    <div class="row">
        <div class="col-md-12">
            <form action="{{ url('patients') }}" method="GET">
                <div class="row">
                    <div class="form-group col-md-10">
                        <input type="text" name="search" class="form-control"
                            placeholder="Digite o nome do paciente">
                    </div>
                    <div class="form-group col-md-2">
                        <button type="submit" class="btn btn-sm btn-success"><i
                                class="tim-icons icon-zoom-split"></i></button>
                        <a href="{{ url('patients') }}" class="btn btn-sm btn-danger"><i
                                class="tim-icons icon-refresh-01"></i></a>
                    </div>
                </div>
            </form>
        </div>
    </div>
</div>
```

Elaborado pela autora

Esse código cria uma barra de busca na página de pacientes usando *HTML*, recursos do *Bootstrap* e *JavaScript*, a barra de busca é composta por um campo de texto onde os usuários podem inserir o nome do paciente que desejam encontrar, essa barra é seguida por dois botões: um botão verde de pesquisa e um botão vermelho de reset. A utilização de ícones no lugar do texto nos botões adiciona uma melhoria visual e estética à interface do usuário.

Quando o usuário digita letras no campo de texto, a busca é feita automaticamente e os registros da tabela são filtrados de acordo com os nomes que coincidem com a busca. Quando o usuário digita o nome completo do paciente e clica no botão de pesquisa, é feita uma requisição ao *Controller* para buscar o nome do paciente.

Quando o evento “*input*” é acionado no campo de busca (*input*[*name*=“*search*”]), o *script* obtém o valor digitado pelo usuário (*searchValue*), o *script*, então, seleciona todas as linhas de pacientes da tabela e percorre cada uma delas, para cada linha, é obtido o nome do paciente da primeira coluna e é

convertido para letras minúsculas para comparação. Se o nome do paciente incluir o valor da busca, a linha é exibida, caso contrário, a linha é ocultada (*display*: *'none'*).

Dessa forma, ao digitar no campo de busca, os pacientes que não correspondem à busca são ocultados da tabela, enquanto os pacientes cujos nomes correspondem são exibidos dinamicamente em tempo real, tornando a busca mais eficiente e interativa para o usuário.

Logo abaixo é apresentado o cabeçalho da tabela que exibe os registros de pacientes. Esse cabeçalho é geralmente utilizado para identificar as colunas da tabela e proporcionar uma melhor organização visual dos dados apresentados. Dentro do corpo do *card*, é utilizada a classe “*table-responsive*” para tornar a tabela responsiva, ajustando-se adequadamente em diferentes tamanhos de tela. A tabela em si é definida com a classe “*tablesorter*” e possui um ID “*table_patients*” para identificação. Em cada coluna do cabeçalho, temos os títulos das informações dos pacientes, como “Nome”, “CPF”, “Data de nascimento”, “Cidade”, “Estado” e “Primeira consulta”, ada título está centralizado (*class*=“*text-center*”) para melhor alinhamento visual.

Na Figura 15, é representado o corpo da tabela que contém os dados dos pacientes, esses dados são carregados dinamicamente pelo Controller da tela.

Figura 15 – Corpo tabela – Tela Paciente

```
<tbody>
    @foreach ($patient as $patients)
        <tr>
            <td class="text-center">
                {{ $patients->nome }}
            </td>
            <td class="text-center">
                {{ $patients->cpf }}
            </td>
            <td class="text-center">
                {{ date('d/m/Y', strtotime($patients->dataNascimento)) }}
            </td>
            <td class="text-center">
                {{ $patients->cidade }}
            </td>
            <td class="text-center">
                {{ $patients->estado }}
            </td>
            <td class="text-center">
                {{ date('d/m/Y', strtotime($patients->dataPrimeiraConsulta)) }}
            </td>
            <td class="text-center">
                <div class="dropdown">
                    <a class="btn btn-sm btn-dark text-white" href="#" role="button"
                        data-toggle="dropdown" aria-haspopup="true" aria-expanded="false">
                        <i class="fas fa-ellipsis-v"></i>
                    </a>
                    <div class="dropdown-menu dropdown-menu-right dropdown-menu-arrow">
                        <a class="dropdown-item"
                            href="{{ url("patients/$patients->id") }}">Visualizar</a>
                        <a class="dropdown-item" href="#">Prontuário</a>
                    </div>
                </div>
            </td>
        </tr>
    @endforeach
</tbody>
```

Elaborado pela autora

Dentro do elemento “*tbody*”, é realizado um *loop* “*foreach*” para iterar sobre os registros dos pacientes que foram passados para a *view* por meio do Controller. Para cada paciente na lista de pacientes, é criada uma linha na tabela onde são exibidos os dados do paciente nas colunas correspondentes. Uma das colunas é o “Ações”, há um botão representado por um ícone vertical (i *class*=“*fas fa-ellipsis*-v”) que, quando clicado, abre um *dropdown* com opções permitindo visualizar os detalhes do paciente e acessar o prontuário associado a ele.

Cada linha da tabela representa um paciente específico e suas informações são exibidas nas células correspondentes. Assim, essa estrutura permite exibir os dados dos pacientes de forma organizada e funcional, tornando a tabela interativa e permitindo ao usuário realizar ações relacionadas aos pacientes diretamente na página.

Cada etapa do *CRUD* possui uma função correspondente no *Controller*, responsável por realizar as operações relacionadas aos dados. A função “*index*” da tela, é uma dessas funções e tem como objetivo exibir os registros da tabela na interface. O método é responsável por exibir uma lista dos registros relacionados aos pacientes. Antes de abordar o método “*index*”, no construtor do Controller, é criada uma instância do modelo “*Patients*” em uma propriedade chamada “$*objPatient*”, isso possibilita o acesso aos métodos e atributos do modelo dentro do Controller, permitindo a interação com os dados relacionados aos pacientes.

Agora, focando na função “*index*”, essa função recebe um parâmetro do tipo “*Request*”, que é utilizado para obter informações enviadas pelo cliente, como os dados de busca na barra de pesquisa. É obtido o valor do parâmetro “*search*” do *Request*, que representa o termo de busca inserido pelo usuário na barra de pesquisa.

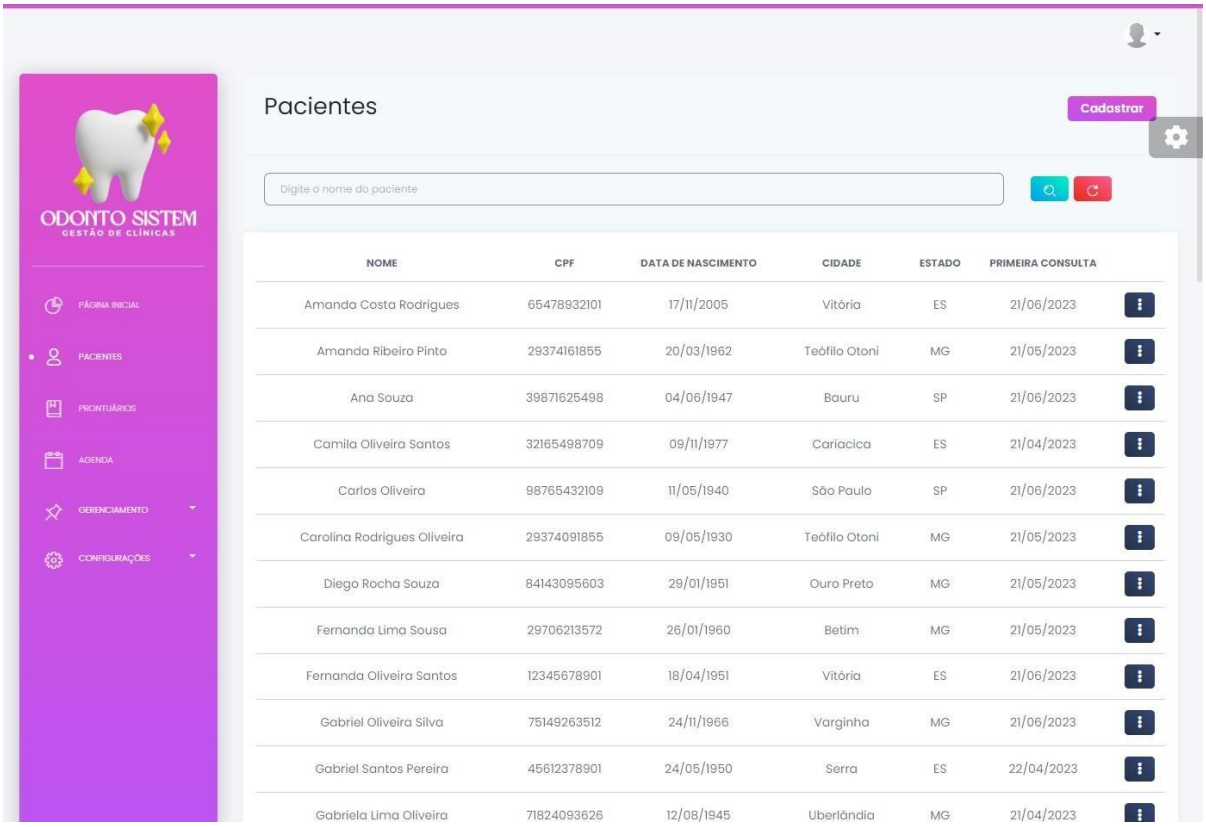
Se existir um valor em “*search*” (ou seja, o usuário fez uma busca), é realizada uma consulta no banco de dados utilizando o modelo “*Patients*” para obter os pacientes cujo nome seja semelhante ao termo de busca. Essa consulta é realizada através do método “*where*” com a cláusula “*like*” para buscar registros que contenham o termo buscado em qualquer parte do nome. Os resultados são ordenados pelo nome do paciente em ordem alfabética utilizando o método “*sortBy*”. Caso não haja um valor em “*search*” (nenhuma busca foi feita), todos os registros de pacientes são obtidos do banco de dados e são ordenados pelo nome.

Por fim, a função retorna a *view* “*pages.patients*”, passando a variável “$*patient*” que contém a lista de pacientes obtida através da busca ou todos os registros, e a variável “$*search*”, que contém o termo de busca digitado pelo usuário. Essas variáveis são passadas para a *view* para que os dados possam ser exibidos na tabela e a barra de busca seja preenchida com o termo buscado anteriormente, se houver.

A Figura 16 apresenta a tela de pacientes, uma página dedicada à gestão e visualização dos pacientes cadastrados na clínica. No topo da tela, há um botão de cadastro que permite adicionar novos pacientes à lista e logo abaixo do botão de cadastro, encontra-se uma barra de busca, que possibilita a pesquisa de pacientes.

A tabela apresentada abaixo da barra de busca exibe todos os pacientes cadastrados. Cada linha da tabela representa um paciente e contém informações importantes, como nome, CPF, data de nascimento, cidade, estado e data da 121 primeira consulta.

Figura 16 – Tela Pacientes




Elaborado pela autora

### 3.3.3.2. Visualização de registro

É importante mencionar que ao selecionar a opção “Visualizar” no *dropdown* de um paciente específico, o usuário é redirecionado para uma tela semelhante à tela de cadastro, porém com os campos bloqueados para edição. Essa funcionalidade permite ao usuário ver todos os detalhes do paciente em uma visualização mais detalhada, sem a possibilidade de realizar alterações acidentais, os campos na tela de visualização são preenchidos com os dados do paciente selecionado, tornando possível a análise completa das informações.

Na Figura 17, é demonstrada a criação dessa tela de visualização de registro de paciente. Essa tela é similar à tela de cadastro, porém, a diferença está na configuração dos campos, que são apresentados apenas para leitura (*disabled*) e não permitem alterações.

Figura 17 – Tela de visualização – Paciente

```
<div class="card">
    <div class="card-header">
        <h5 class="title">{{ _('Dados Pessoais') }}</h5>
    </div>
    <form>
        @csrf
        <div class="card-body">
            <div class="row">
                <div class="form-group col-md-8">
                    <label>{{ _('Nome') }}</label>
                    <input type="text" name="nome" class="form-control" value="{{ $patient->nome }}"
                        disabled>
                </div>
                <div class="form-group col-md-4">
                    <label>{{ _('CPF') }}</label>
                    <input type="text" name="cpf" class="form-control" value="{{ $patient->cpf }}"
                        disabled>
                </div>
            </div>
            <div class="row">
                <div class="form-group col-md-6">
                    <label>{{ _('Data de Nascimento') }}</label>
                    <input type="date" name="dataNascimento" class="form-control"
                        value={{ date($patient->dataNascimento) }} disabled>
                </div>
                <div class="form-group col-md-6">
                    <label>{{ _('Primeira Consulta') }}</label>
                    <input type="date" name="dataPrimeiraConsulta" class="form-control"
                        value={{ date($patient->dataPrimeiraConsulta) }} disabled>
                </div>
            </div>
            <div class="row">
                <div class="form-group col-md-12">
                    <label>{{ _('Rua') }}</label>
                    <input type="text" name="rua" class="form-control" value="{{ $patient->rua }}"
                        disabled>
                </div>
            </div>
```

Elaborado pela autora

No código é utilizado um *card* para organizar visualmente o conteúdo, o cabeçalho exibe o título “Dados Pessoais”, enquanto o corpo contém um formulário para exibir os campos de input que representam os dados do paciente. Cada campo é acompanhado de um rótulo descritivo e são definidos como somente leitura (*disabled*), impedindo a edição direta dos dados nessa tela. O valor de cada campo

é preenchido dinamicamente com os dados do paciente correspondente através dos dados retornados pelo *Controller*.

O código repete essa estrutura para todos os dados do paciente, permitindo que todas as informações sejam visualizadas sem possibilidade de alteração, dessa forma, a tela proporciona uma visualização detalhada dos dados do paciente em um formato fácil de ler e interagir, facilitando a análise completa das informações do paciente selecionado.

Após a exibição dos dados do paciente na tela de visualização, são apresentados dois botões que permitem realizar ações específicas. O botão “Voltar” redireciona o usuário para a página de listagem de pacientes, permitindo que ele retorne à lista após visualizar os detalhes de um paciente específico. Já o botão “Editar” leva o usuário para a página de edição do paciente selecionado, possibilitando que ele faça alterações nos dados cadastrados. Cada botão é estilizado com as classes do *Bootstrap* para apresentar um tamanho reduzido e cores apropriadas para ações específicas (como *btn-primary* para “Voltar” e *btn-success* para “Editar”).

A função “*show*” do *Controller* tem como objetivo exibir os detalhes de um paciente específico com base no ID fornecido como parâmetro. Quando o usuário seleciona a opção “Visualizar” para um paciente na lista, o sistema redireciona para essa função, que busca as informações do paciente no banco de dados utilizando o ID fornecido, em seguida, a função carrega a *view* “*patients*.*show*” e passa os dados do paciente como uma variável chamada “*patient*”, permitindo que esses dados sejam acessados e exibidos na tela de visualização.

Essa função é responsável por fornecer os detalhes do paciente de forma individualizada, garantindo que o usuário possa acessar informações específicas sobre cada registro no sistema. Os resultados dessa tela podem ser encontrados no Capítulo 4, na sessão que aborda a Análise dos Resultados e a Avaliação do Sistema.

#### 3.3.3.3. Cadastro ou edição de registro

A tela “*create*.*blade*.*php*” segue o mesmo padrão da tela de visualização, apresentando uma estrutura similar para edição e a inserção de novos dados de pacientes. O fato de a mesma tela permitir tanto a edição quanto o cadastro de novos pacientes dependerá da chamada realizada a partir das rotas e dos dados retornados na *URL*, a Figura 18 representa a criação dessa tela.

Figura 18 – Tela create – Paciente

```
<div class="card">
    <div class="card-header">
        <h5 class="title">
            @if (isset($patient))
                Editar Paciente
            @else
                Cadastrar Paciente
            @endif
        </h5>
    </div>

    @if (isset($patient))
        <form name="formEditPatient" id="formEditPatient" method="post"
            action="{{ url("patients/$patient->id") }}">
            @method('PUT')
        @else
            <form name="formCadPatient" id="formCadPatient" method="post" action="{{ url('patients') }}">
    @endif
    @csrf
```

Elaborado pela autora

O cabeçalho do *card* exibe o título “Cadastrar Paciente” se não houver informações do paciente ($*patient*) definidas, indicando que a tela é utilizada para inserir um novo paciente. Caso contrário, se houver informações do paciente, o título é alterado para “Editar Paciente”, sinalizando que a tela será utilizada para modificar os dados de um paciente existente.

A seguir, o código verifica novamente se há informações do paciente definidas, se houver, é criado um formulário para a edição dos dados do paciente, onde a ação do formulário aponta para a rota de atualização do paciente específico. Além disso, é incluído o método “@*method*(*'PUT'*)”, indicando que a requisição *HTTP* para a rota será do tipo *PUT*, comumente utilizado para atualizar registros no servidor.

Caso não haja informações do paciente definidas, é criado um formulário para o cadastro de novos pacientes, onde a ação do formulário aponta para a rota de criação de novos pacientes (*url*(*'patients'*)). O uso da diretiva "*@csrf*" garante que o formulário tenha o *token CSRF* necessário para proteção contra ataques de falsificação de solicitações entre sites.

Dessa forma, o código flexibiliza a mesma tela para a criação ou edição de pacientes, proporcionando uma experiência mais conveniente e organizada para o usuário, dependendo das informações passadas e das rotas acessadas.

Na Figura 19, é construído o formulário com todos os campos necessários para cadastrar ou editar os dados de um paciente.

Figura 19 – Corpo do formulário – Paciente

```
<div class="card-body">
    <div class="row">
        <div class="form-group col-md-8">
            <label>{{ _('Nome') }}</label>
            <input type="text" name="nome" class="form-control" value="{{ $patient->nome ?? '' }}"
                placeholder="Inserir nome completo" required>
        </div>
        <div class="form-group col-md-4">
            <label>{{ _('CPF') }}</label>
            <input type="text" name="cpf" class="form-control" value="{{ $patient->cpf ?? '' }}"
                placeholder="Inserir apenas números" required>
        </div>
    </div>
    <div class="row">
        <div class="form-group col-md-6">
            <label>{{ _('Data de Nascimento') }}</label>
            <input type="date" name="dataNascimento" class="form-control"
                value={{ date($patient->dataNascimento ?? '') }} required>
        </div>
        <div class="form-group col-md-6">
            <label>{{ _('Primeira Consulta') }}</label>
            <input type="datetime-local" name="dataPrimeiraConsulta" class="form-control"
                value="{{ isset($patient) ? date('Y-m-d\TH:i', strtotime($patient->dataPrimeiraConsulta)) : '' }}"
                required>
        </div>
    </div>
```

Elaborado pela autora

O trecho de código apresenta a construção do formulário de cadastro ou edição de pacientes. O formulário é organizado em colunas para melhor disposição dos campos de *input*, cada campo é representado por uma *div*, onde indica o tamanho da coluna ocupado pelo campo. O formulário possui os campos essenciais para registrar informações sobre o paciente, para cada campo, há um rótulo descritivo e um campo de *input* para inserção das informações.

Os campos são preenchidos com os dados do paciente caso o formulário esteja sendo acessado para edição, caso contrário, os campos estarão vazios, prontos para receber novos dados, além disso, é definida a propriedade *required* para alguns campos, garantindo que essas informações sejam obrigatórias no momento do envio do formulário. Essa estrutura do formulário facilita a coleta e atualização das informações do paciente de forma organizada e eficiente.

Os campos “Cidade” e “Estado” do formulário de cadastro ou edição de pacientes, possuem uma estrutura um pouco diferente dos demais. O campo “Cidade” é uma lista suspensa representada por um elemento “<*select*>“, que será preenchido dinamicamente com as opções de cidades associadas ao estado selecionado, isso é feito através de um script que faz uma requisição à *API* do *IBGE* para obter as cidades do estado selecionado no campo “Estado”.

O campo “Estado” também é uma lista suspensa representada por um elemento “<*select*>“, ele possui uma opção padrão “Selecione um estado” para permitir uma seleção inicial. As opções de estado são geradas a partir de um *loop* “*@foreach*”, que itera sobre um *array* de estados, cada estado é representado por um elemento “<*option*>“ que exibe o nome completo do estado e tem um valor igual ao seu nome.

Essa implementação garante que o usuário possa selecionar o estado desejado, e em seguida, o campo “Cidade” será atualizado dinamicamente para exibir apenas as opções de cidades relacionadas a esse estado, proporcionando uma experiência mais fluída e intuitiva no cadastro ou edição de pacientes.

Os dados desses dois campos são carregados dinamicamente no formulário de cadastro ou edição de pacientes. Através de um script em *JavaScript*, o comportamento do campo “Cidade” é adaptado de acordo com o estado selecionado no campo “Estado”.

Primeiro, o *script* seleciona os elementos *HTML* que representam os campos “Estado” e “Cidade” por meio do método “*document*.*querySelector*()”, em seguida, o código define um objeto chamado “estadosSiglas”, que faz o mapeamento entre o nome completo dos estados e suas respectivas siglas. A função “getSiglaEstado()” é definida para obter a sigla do estado selecionado no campo “Estado” usando a

propriedade “*value*” do elemento *HTML* selecionado. A função retorna a sigla correspondente ao estado selecionado usando o objeto “estadosSiglas”.

A função “*updateCidades*()” é responsável por essa atualização, quando o usuário seleciona um estado no campo “Estado”, a função é acionada. Ela começa obtendo a sigla do estado selecionado usando a função “*getSiglaEstado*()”, em seguida, o script faz uma requisição usando o método “*fetch*()” para a *API* pública do IBGE que fornece as informações sobre as cidades brasileiras. A *URL* da *API* é construída usando a sigla do estado obtida anteriormente, a resposta da *API* é tratada como um objeto *JSON*.

Após obter os dados das cidades, o script limpa as opções existentes no campo “Cidade” usando “*cidadeSelect.innerHTML*”, em seguida, adiciona uma opção padrão “Selecione uma cidade” usando o método “*createElement*()” e “*appendChild*()”. O script então itera sobre as cidades obtidas e cria novas opções para cada uma delas, utilizando o método “*createElement*()” para criar um elemento “*option*” e adicioná-lo ao campo “Cidade” usando o método “*appendChild*()”, além disso, o *script* verifica se existe um paciente em edição (*isset*($*patient*)) e, se houver, define a cidade correspondente ao paciente existente como a opção selecionada no campo “Cidade”.

Por fim, o evento “*change*” é adicionado ao campo “Estado” *(estadoSelect.addEventListener('change', updateCidades))* para que sempre que o estado for alterado, a função “*updateCidades*()” seja chamada para atualizar as opções do campo “Cidade” com base no estado selecionado, dessa forma, o usuário tem acesso apenas às cidades associadas ao estado escolhido.

As funções “*showMessage(message)”* e ‘*hideMessage*()” são utilizadas para exibir e ocultar a mensagem de erro, respectivamente, a mensagem de erro é mostrada por meio de uma “*div*” que é estilizada com as classes “*alert*” e “*alert-danger*”, proporcionando uma aparência de destaque visual para a mensagem de erro. Essas funcionalidades adicionais ajudam a melhorar a usabilidade do formulário, fornecendo *feedback* ao usuário sobre a necessidade de selecionar um estado antes de escolher a cidade e fornecendo uma maneira de ocultar a mensagem de erro quando o usuário realiza a seleção correta.

Na tela de cadastro e edição de pacientes, a parte inferior é composta pelos botões essenciais para a interação do usuário com o sistema. Os botões são apresentados dentro de uma “*div*” com a classe “*card-footer*”, o que os posiciona de forma centralizada na página. O primeiro botão é o “Voltar”, e sua função varia dependendo do contexto: se o usuário estiver editando um paciente existente, o botão o redirecionará para a página de visualização desse paciente em específico, caso esteja criando um paciente, o botão o levará de volta à lista geral de pacientes. Em seguida, temos o botão “Salvar”, que é utilizado para registrar as informações inseridas ou atualizadas no formulário. Por fim, caso a tela esteja no modo de edição de paciente, é disponibilizado o botão “Deletar”, que permite a exclusão do registro do paciente.

A função “*create*” no *Controller* é responsável por exibir o formulário de criação de um novo registro de paciente, como demonstrado na Figura 20.

Figura 20 – Controller de criação

```php
/**
 * Show the form for creating a new resource.
 *
 * @return \Illuminate\Http\Response
 */
public function create()
{
    $response = Http::withOptions(['verify' => false])->get('https://servicodados.ibge.gov.br/api/v1/localidades/estados');
    $estados = $response->json();

    return view('patients.create')->with('estados', $estados);
}

/**
 * Store a newly created resource in storage.
 *
 * @param  \Illuminate\Http\Request  $request
 * @return \Illuminate\Http\Response
 */
public function store(Request $request)
{
    $cadastro = $this->objPatient->create([
        'nome' => $request->nome,
        'cpf' => $request->cpf,
        'dataNascimento' => $request->dataNascimento,
        'cep' => $request->cep,
        'rua' => $request->rua,
        'bairro' => $request->bairro,
        'numero' => $request->numero,
        'cidade' => $request->cidade,
        'estado' => $request->estado,
        'dataPrimeiraConsulta' => $request->dataPrimeiraConsulta
    ]);
    if ($cadastro) return redirect('patients');
}
```

Elaborado pela autora

Quando o usuário acessa a página de cadastro de pacientes, essa função é chamada e realiza uma requisição *HTTP* para a *API* do *IBGE*, obtendo a lista de estados brasileiros. Essa lista é então passada para a *view* "*patients*.create" através da variável "$estados", permitindo que os estados sejam exibidos no campo de seleção correspondente no formulário de cadastro.

Por outro lado, a função "*store*" é responsável por salvar o novo registro de paciente no banco de dados. Ao submeter o formulário de cadastro, os dados inseridos pelo usuário são capturados através da instância da classe "*Request*", em seguida, esses dados são utilizados para criar um novo registro de paciente no banco de dados, utilizando o método "*create*" do modelo "*Patients*". Após o cadastro ser realizado com sucesso, o sistema redireciona o usuário de volta para a página de listagem de pacientes, garantindo que o novo registro seja exibido na lista atualizada.

Essas duas funções trabalham em conjunto para possibilitar a criação e o armazenamento de novos registros de pacientes no sistema, fornecendo uma interface para o usuário inserir informações sobre o paciente e gerenciar esses dados de forma organizada e eficiente.

As funções "*edit*" e "*update*" no *Controller* são responsáveis por permitir a edição de um registro de paciente existente, demonstrado na Figura 21.

Figura 21 – Controller de edição

```php
/**
 * Show the form for editing the specified resource.
 *
 * @param  int  $id
 * @return \Illuminate\Http\Response
 */
public function edit($id)
{
    $response = Http::withOptions(['verify' => false])->get('https://servicodados.ibge.gov.br/api/v1/localidades/estados');
    $estados = $response->json();

    $patient = $this->objPatient->find($id);

    return view('patients.create')->with('estados', $estados)->with('patient', $patient);
}

/**
 * Update the specified resource in storage.
 *
 * @param  \Illuminate\Http\Request  $request
 * @param  int  $id
 * @return \Illuminate\Http\Response
 */
public function update(Request $request, $id)
{
    $this->objPatient->where(['id' => $id])->update([
        'nome' => $request->nome,
        'cpf' => $request->cpf,
        'dataNascimento' => $request->dataNascimento,
        'cep' => $request->cep,
        'rua' => $request->rua,
        'bairro' => $request->bairro,
        'numero' => $request->numero,
        'cidade' => $request->cidade,
        'estado' => $request->estado,
        'dataPrimeiraConsulta' => $request->dataPrimeiraConsulta
    ]);
    return redirect('patients');
}
```

Elaborado pela autora

A função “*edit*” recebe o *ID* do paciente como parâmetro e, a partir desse *ID*, busca o registro do paciente no banco de dados utilizando o método “*find*” do modelo “*Patients*”, além disso, realiza uma requisição *HTTP* para a *API* do *IBGE* para obter a lista de estados, que será utilizada para popular o campo de seleção de estados no formulário de edição. Em seguida, a função retorna a *view* “*patients.create*” juntamente com os dados do paciente a serem editados e a lista de estados.

Já a função “*update*” é acionada quando o usuário envia o formulário de edição. Os dados inseridos no formulário são capturados através da instância da classe “*Request*” e são utilizados para atualizar o registro do paciente no banco de dados, utilizando o método “*update*” do modelo “*Patients*”, após a atualização ser

realizada com sucesso, o sistema redireciona o usuário de volta para a página de listagem de pacientes, mostrando a lista atualizada com as alterações realizadas.

Essas duas funções permitem que os dados dos pacientes sejam editados de forma simples e segura, possibilitando ao usuário manter os registros atualizados e corretos no sistema. A utilização da *API* do *IBGE* para obter a lista de estados torna o processo de edição mais eficiente, facilitando a seleção do estado correto para o paciente.

Os resultados dessa tela podem ser encontrados no Capítulo 4, na sessão que aborda a Análise dos Resultados e a Avaliação do Sistema.

#### 3.3.3.4. Exclusão de um registro

A função "*delete'* (ou "*destroy*") no *CRUD* é de extrema importância, pois ela permite a remoção de registros do banco de dados, isso é fundamental para manter a integridade dos dados e garantir que informações indesejadas ou desnecessárias não permaneçam no sistema.

A função recebe o parâmetro "$*id*", que representa o identificador único do registro que se deseja excluir. Dentro da função, é feita a chamada ao método "*destroy*" do modelo "*Patients*", passando o "$*id*" como argumento, esse método é responsável por realizar a exclusão do registro do banco de dados. Se a operação de exclusão for bem-sucedida, o método "*destroy*" retorna verdadeiro (*true*), caso contrário, retorna falso (*false*).

O trecho de código *JavaScript* da Figura 22 e 23 é responsável por lidar com a exclusão de registros no *front-end* do aplicativo.

Figura 22 – JavaScript de exclusão

```
//Delete
function confirmDel(event){
    event.preventDefault();
    console.log(event.target.parentNode.href);

    let token = doc.getElementsByName("_token")[0].value;

    if(confirm("Deseja mesmo apagar?")){
        let ajax = new XMLHttpRequest();
        ajax.open("DELETE", event.target.parentNode.href);
        ajax.setRequestHeader('X-CSRF-TOKEN', token);
        ajax.onreadystatechange = function(){
            if(ajax.readyState === 4 && ajax.status === 200){
                if(doc.querySelector('.del-patients')) win.location.href="/patients";
                if(doc.querySelector('.del-clinicals')) win.location.href="/clinicals";
                if(doc.querySelector('.del-dentists')) win.location.href="/dentists";
                if(doc.querySelector('.del-handbooks')) win.location.href="/handbooks";
                if(doc.querySelector('.del-users')) win.location.href="/user";
            }
        };
        ajax.send();
    } else{
        return false;
    }
}
```

Elaborado pela autora

Esse trecho de código *JavaScript* é responsável por lidar com a exclusão de registros no *front-end* do aplicativo. Ele implementa a função “*confirmDel*” que é acionada quando o usuário deseja excluir um registro, o *script* verifica se o usuário realmente deseja executar a exclusão, exibindo um prompt de confirmação. Dentro da função “*confirmDel*”, é feita uma requisição *AJAX* do tipo “*DELETE*” para a *URL* especificada nos atributos “*href*” dos botões de exclusão. Essa *URL* corresponde à rota no servidor responsável por efetuar a exclusão do registro no banco de dados.

Antes de enviar a requisição, o código obtém o valor do *token CSRF* (*Cross-Site Request Forgery*) da página, que é adicionado como cabeçalho na requisição para proteger contra ataques *CSRF*. Quando a requisição *AJAX* é concluída com sucesso (*status* 200), o código redireciona o usuário para a página correspondente, dependendo do tipo de registro excluído, por exemplo, se um paciente for excluído, o redirecionamento levará o usuário de volta à página de listagem de pacientes (“/*patients*”).

Figura 23 – JavaScript de exclusão

```
if(doc.querySelector('.del-patients')){
    let btn = doc.querySelectorAll('.del-patients');
    for(let i=0; i<btn.length; i++){
        btn[i].addEventListener('click', confirmDel, false);
    }
}

if(doc.querySelector('.del-clinicals')){
    let btn = doc.querySelectorAll('.del-clinicals');
    for(let i=0; i<btn.length; i++){
        btn[i].addEventListener('click', confirmDel, false);
    }
}

if(doc.querySelector('.del-dentists')){
    let btn = doc.querySelectorAll('.del-dentists');
    for(let i=0; i<btn.length; i++){
        btn[i].addEventListener('click', confirmDel, false);
    }
}

if(doc.querySelector('.del-handbooks')){
    let btn = doc.querySelectorAll('.del-handbooks');
    for(let i=0; i<btn.length; i++){
        btn[i].addEventListener('click', confirmDel, false);
    }
}

if(doc.querySelector('.del-users')){
    let btn = doc.querySelectorAll('.del-users');
    for(let i=0; i<btn.length; i++){
        btn[i].addEventListener('click', confirmDel, false);
    }
}
```

Elaborado pela autora

O evento de clique nos botões de exclusão (identificados por classes como “*del-patients*”, “*del-clinicals*”, “*del-dentists*”, *“del-handbooks*” e “*del-users*”) é escutado e, ao ser acionado, chama a função “*confirmDel*”. Esses botões são responsáveis por acionar a exclusão de registros relacionados a pacientes, clínicos, dentistas, prontuários e usuários, respectivamente.

Em resumo, esse trecho de código *JavaScript* adiciona funcionalidade aos botões de exclusão do *front-end*, permitindo que o usuário confirme a exclusão antes

de enviar a requisição de exclusão para o servidor, e em seguida, redireciona o usuário para a página correta após a exclusão. Essa abordagem proporciona uma interação mais amigável com o usuário e evita a exclusão acidental de registros importantes.

#### 3.3.3.5. Model e Rotas CRUD

O Model é uma das três partes principais do *MVC* e representa a camada de dados ou a camada de negócios da aplicação. Ele é responsável por lidar com a lógica de negócios, a interação com o banco de dados e o tratamento dos dados que serão utilizados pela aplicação.

Em um contexto *Laravel* (ou qualquer outro *framework* que siga o padrão *MVC*), os *Models* são classes que representam entidades do domínio da aplicação, como pacientes, clínicas, dentistas, dentre outros. Eles definem a estrutura dos dados, os relacionamentos com outras entidades e as regras de negócio relacionadas a essas entidades. O código apresentado na Figura 24 representa um exemplo de um *Model* no *framework Laravel*, especificamente para a tabela "*Patients*".

Figura 24 – Model Patients

```
class Patients extends Model
{
    protected $table='patients';
    protected $fillable=['nome', 'cpf', 'dataNascimento', 'dataPrimeiraConsulta', 'rua', 'bairro', 'numero', 'cep', 'cidade', 'estado'];

    public function relHandbooks(){
        return $this->hasMany('App\Models\Handbooks', 'id_paciente');
    }

    public function relSchedule(){
        return $this->hasMany('App\Models\Schedule', 'id_paciente');
    }
}
```

Elaborado pela autora

A propriedade "$*table*" especifica o nome da tabela no banco de dados que corresponde aos registros de pacientes. A propriedade "$*fillable*" define quais colunas da tabela podem ser preenchidas, permitindo atribuições, isso é útil para operações como criação e atualização de registros. Além disso, o *Model* inclui dois métodos de relacionamento: "*relHandbooks*" e "*relSchedule*", esses métodos

definem os relacionamentos entre pacientes e outras entidades do sistema, como prontuários e agendamentos, dessa forma, o Model facilita a interação com o banco de dados, permitindo consultas, inserções, atualizações e exclusões de forma mais simples e intuitiva.

Ele também fornece mecanismos para acessar dados relacionados a partir de um paciente específico, melhorando a organização e eficiência do código. Com essa estrutura, o *Model* desempenha um papel fundamental no processo do *CRUD*, garantindo a integridade e manipulação dos dados das tabelas do banco.

As rotas são partes fundamentais de um aplicativo *Laravel*, pois são responsáveis por direcionar as solicitações *HTTP* para as ações adequadas nos controladores. Na Figura 25, temos um trecho de código que define algumas rotas relacionadas ao *CRUD* do aplicativo.

Figura 25 – Rotas web.php

```
Route::group(['middleware' => 'auth'], function () {
    Route::resource('patients', 'App\Http\Controllers\PatientsController');
    Route::resource('clinicals', 'App\Http\Controllers\ClinicalsController');
    Route::resource('dentists', 'App\Http\Controllers\DentistsController');
    Route::resource('user', 'App\Http\Controllers\UserController');
    Route::resource('handbooks', 'App\Http\Controllers\HandbooksController');
    Route::get('schedule', [App\Http\Controllers\ScheduleController::class, 'index']);
    Route::post('schedule/store', [App\Http\Controllers\ScheduleController::class, 'store']);
    Route::get('schedule/load-events', 'App\Http\Controllers\EventController@loadEvents')->name('routeLoadEvents');
    Route::put('schedule/event-update', 'App\Http\Controllers\EventController@update')->name('routeEventUpdate');
    Route::post('schedule/event-store', 'App\Http\Controllers\EventController@store')->name('routeEventStore');
    Route::delete('schedule/event-delete', 'App\Http\Controllers\EventController@destroy')->name('routeEventDelete');
    Route::get('schedule/check-duplicate-event/{start?}/{end?}/{id_dentista?}', 'App\Http\Controllers\EventController@checkDuplicateEvent')
    ->name('routeCheckDuplicateEvent');
    Route::get('schedule/get-patients/{id?}', 'App\Http\Controllers\EventController@getPatients')->name('routeGetPatients');
    Route::get('schedule/get-dentists/{id?}', 'App\Http\Controllers\EventController@getDentists')->name('routeGetDentists');
    Route::get('schedule/get-clinicals/{id?}', 'App\Http\Controllers\EventController@getClinicals')->name('routeGetClinicals');
    Route::get('/estados', 'EstadoController@index');
    Route::get('/consultas-por-mes', 'App\Http\Controllers\ConsultaController@consultasPorMes');
    Route::get('/consultas-por-ano', 'App\Http\Controllers\ConsultaController@consultasPorAno');
    Route::get('/pacientes-por-estado', 'App\Http\Controllers\ConsultaController@pacientesPorEstado');
});
```

Elaborado pela autora

As rotas são mapeamentos entre *URLs* e métodos dos *Controllers*, permitindo que o sistema responda a diferentes requisições de acordo com o caminho acessado pelo usuário. As rotas definidas com o método “*Route::resource*” estão relacionadas ao *CRUD* das entidades do sistema, como pacientes, prontuários, dentistas, usuários e clínicas, essas rotas geram automaticamente as rotas *RESTful* padrão para cada uma dessas entidades, incluindo as rotas para listar, criar, editar e excluir registros.

Além disso, existem outras rotas personalizadas, como as relacionadas ao agendamento de consultas (*schedule*), que possuem ações específicas para criar, atualizar e excluir eventos no calendário. Também há rotas para obter pacientes, dentistas e clínicas disponíveis para agendamento, bem como outras rotas que fornecem dados estatísticos sobre as consultas e pacientes cadastrados.

Cada rota é mapeada para o *Controller* responsável por processar a requisição e retornar a resposta adequada. Essas rotas permitem que o sistema funcione de forma organizada, garantindo o correto processamento das ações do usuário e a integração com o banco de dados e outras funcionalidades do sistema.

### 3.3.4. Calendário

O calendário criado para fazer agendamentos na página de agenda apresenta uma abordagem interativa e amigável ao usuário, permitindo que ele clique sobre um dia específico para criar uma agenda ou visualize os agendamentos já cadastrados. Essa interatividade é obtida através do uso do *FullCalendar*, uma biblioteca JavaScript popular para criação de calendários interativos.

Ao utilizar o *FullCalendar*, é possível implementar diversas funcionalidades, como cadastrar, excluir e editar agendamentos diretamente no calendário, além disso, o usuário pode arrastar e modificar a data e hora de um agendamento já existente, tornando o processo de gerenciamento de agendas mais intuitivo.

A visualização do calendário exibe as datas de forma clara e organizada, facilitando a identificação de horários disponíveis e ocupados. Para os agendamentos já cadastrados, o calendário mostra informações relevantes, como o horário e o nome do paciente associado a cada agenda, proporcionando uma visão rápida e eficiente das atividades programadas.

Essa abordagem de utilizar o *FullCalendar* é vantajosa, pois agrega recursos interativos e dinâmicos à página de agenda, tornando-a mais funcional e agradável para o usuário. Com isso, o processo de agendamento e gerenciamento de consultas se torna mais eficiente e organizado, contribuindo para uma melhor experiência do usuário com o sistema.

#### 3.3.4.1. Criação do calendário

Na Figura 26 é possível observar um trecho do código em *“blade.php*” que é responsável por criar a estrutura do calendário.

Figura 26 – HTML de criação do calendário

```
<div class="row">
    <div class="col-md-12">
        <div class="card">
            <div class="card-body" id="agenda" data-route-load-events="{{ route('routeLoadEvents') }}"
                data-route-event-update="{{ route('routeEventUpdate') }}"
                data-route-event-store="{{ route('routeEventStore') }}"
                data-route-event-delete="{{ route('routeEventDelete') }}"
                data-route-check-duplicate-event="{{ route('routeCheckDuplicateEvent',
                ['start' => $start ?? null, 'end' => $end ?? null, 'id_dentista' => $id_dentista ?? null]) }}">
                {{-- Calendário --}}
            </div>
        </div>
    </div>
</div>
```

Elaborado pela autora

Nesse código, é criada uma divisão que abriga um “*card*”, dentro temos o “*card-body*”, que é o corpo do *card*, onde o calendário será renderizado, nessa seção, são definidos os atributos “*data-route-*”,* que armazenam as rotas utilizadas para realizar diversas operações do calendário, como carregar eventos, atualizar eventos, criar eventos e excluir eventos. Essas rotas são obtidas usando a função “*Route*” do *Laravel*, que mapeia os nomes das rotas para suas *URLs* correspondentes. O trecho de código contém um comentário “<!-- Calendário -->“, indicando que essa é a área reservada para a renderização do calendário em si. Esse calendário é gerado dinamicamente através do *JavaScript*, utilizando a biblioteca *FullCalendar* mencionada anteriormente.

No trecho de código representado pela Figura 27, é criado o calendário utilizando *JavaScript*. A biblioteca *FullCalendar* é utilizada para essa finalidade, permitindo que o calendário seja renderizado e que suas funcionalidades sejam configuradas.

Figura 27 – Calendário – JavaScript

```
document.addEventListener("DOMContentLoaded", function () {
    let formulario = document.querySelector("form");
    var calendarEl = document.getElementById("agenda");
    var calendar = new FullCalendar.Calendar(calendarEl, {
        initialView: "dayGridMonth",

        locale: "pt-br",
        navLinks: true,
        eventLimit: true,
        selectable: true,
        editable: true,
        droppable: true,
        drop: function (arg) {
            if (document.getElementById("drop-remove").checked) {
                arg.draggedEl.parentNode.removeChild(arg.draggedEl);
            }
        },

        headerToolbar: {
            left: "prev, next today",
            center: "title",
            right: "dayGridMonth, timeGridWeek, listWeek",
        },
```

Elaborado pela autora

No trecho de código apresentado, é criado o calendário utilizando a biblioteca *FullCalendar* dentro de uma função anônima que é executada imediatamente após o carregamento do *DOM (DOMContentLoaded*). Dentro da função, é obtido o elemento do formulário através do seletor "*querySelector*", permitindo o acesso aos elementos *HTML* do formulário, em seguida, é recuperado o elemento *HTML* com o ID "agenda" que será utilizado como container para renderizar o calendário, a variável "*calendarEl*" representa esse elemento.

O objeto "*calendar*" é criado usando a classe "*FullCalendar.Calendar*", e várias opções de configuração são definidas para personalizar o calendário. Por exemplo:

- "*initialView*" é definido como "*dayGridMonth*", o que torna a visualização inicial do calendário em modo de mês.

- “*locale*” é configurado para “pt-br”, indicando que o idioma do calendário será em português do Brasil.
- “*navLinks*”, “*eventLimit*”, “*selectable*”, “*editable*” e “*droppable*” são definidos como “*true*” para habilitar as respectivas funcionalidades no calendário, como *links* de navegação, limitação de eventos, seleção, edição e suporte a arrastar e soltar.
- A função “*drop*” é definida para manipular o evento de soltar um evento arrastado no calendário. Neste caso, é verificado se o checkbox com o ID “*drop-remove*” está marcado, e se sim, o evento arrastado é removido do *DOM*.

A propriedade “*headerToolbar*” é utilizada para personalizar a barra de ferramentas do cabeçalho do calendário, especificando os botões que estarão presentes na esquerda, centro e direita do cabeçalho. O objeto de configuração do calendário é finalizado e o calendário é inicializado utilizando a função “*FullCalendar.Calendar*” com os parâmetros “*calendarEl*” e o objeto de configuração “*calendar*”.

#### 3.3.4.2. Select

O trecho de código da Figura 28 e 29 apresenta uma função chamada “*select*” que é acionada quando um evento é selecionado no calendário.

Figura 28 – Calendário – JavaScript

```
select: function (evento) {
    clearMessages("#message");
    resetForm("#formAgenda");
    $("#modalAgenda").modal("show");
    $("#modalAgenda #modalTitle").text("Adicionar Agenda");
    $("#modalAgenda button.deleteEvent").css("display", "none");

    // ID
    id = "";

    // Data Start
    let start = new Date(evento.start);
    let startano = start.getFullYear();
    let startmes = start.getMonth() + 1;
    let startdia = start.getDate();
    let startFormact =
        startano +
        "-" +
        (startmes < 10 ? "0" + startmes : startmes) +
        "-" +
        (startdia < 10 ? "0" + startdia : startdia);
    $("#modalAgenda input[name='start']").val(startFormact);

    // Preencher o campo startTime com 8h
    let startTimeSelect = document.querySelector('select[name="startTime"]');
    startTimeSelect.value = "08:00";
```

Elaborado pela autora

Em resumo, essa função é acionada quando um evento no calendário é selecionado. Ela preenche o formulário de agendamento no *modal* com as informações da data e hora do evento selecionado e exibe o *modal* para permitir que o usuário edite o evento selecionado ou crie um agendamento. Vou explicar o que acontece dentro dessa função:

1. *clearMessages(“#message*”): Essa função é responsável por limpar mensagens de erro que possam ter sido exibidas anteriormente.
2. *resetForm(“#formAgenda”):* Essa função é responsável por resetar o formulário de agendamento, removendo qualquer valor preenchido previamente.
3. *$(“#modalAgenda”).modal(“show”):* Mostra o modal de agendamento, exibindo-o para o usuário.

4. *$(“#modalAgenda #modalTitle”).text(“Adicionar Agenda”):* Define o título do modal como “Adicionar Agenda”.
5. *$(“#modalAgenda button.deleteEvent”).css(“display”, “none”):* Esconde o botão de exclusão de eventos dentro do modal, pois estamos adicionando um novo evento, portanto, não é possível excluir algo que ainda não foi criado.
6. ID = “ “: Define a variável “ID” como uma string vazia, usada para identificar o evento selecionado (nesse caso, não há evento selecionado, está sendo criado um novo).
7. Manipulação das datas de início e fim do evento selecionado:
    - *let start = new Date(evento.start):* Converte a data de início do evento selecionado em um objeto de data.
    - *let startano = start.getFullYear():* Obtém o ano da data de início.
    - *let startmes = start.getMonth() + 1:* Obtém o mês da data de início (+1 porque os meses são indexados de 0 a 11).
    - *let startdia = start.getDate():* Obtém o dia do mês da data de início.
    - *let startFormact = ...:* Cria uma string formatada da data de início no formato “aaaa-mm-dd”.
    - *$(“#modalAgenda input[name='start']”).val(startFormact):* Define o valor do campo de data de início do formulário no modal com a data formatada.
8. Preenchimento do campo de hora de início:
    - Obtém o seletor do campo de hora de início.
    - Define o valor do campo de hora de início como “08:00”.

Figura 29 – Calendário – JavaScript

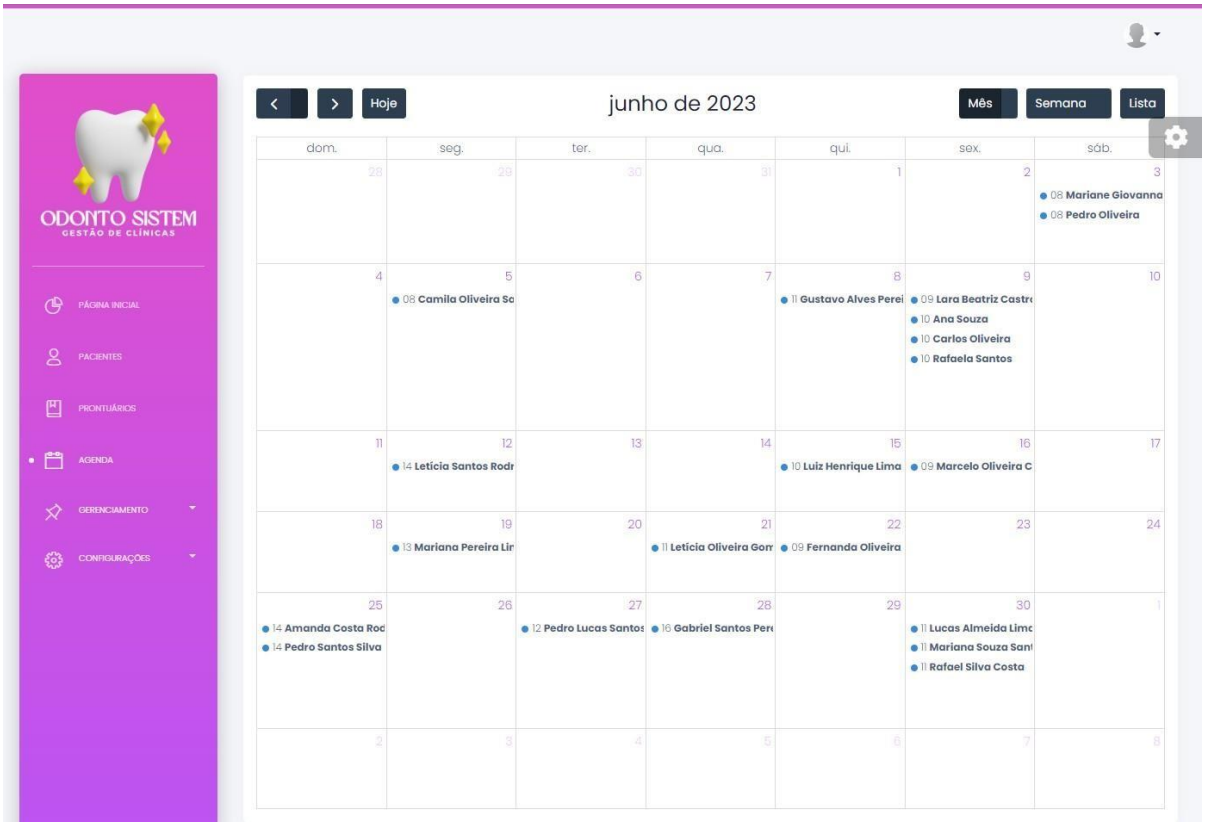
```
// Data End
let end = new Date(evento.end);
let endano = end.getFullYear();
let endmes = end.getMonth() + 1;
let enddia = end.getDate() - 1;
let endFormact =
    endano +
    "-" +
    (endmes < 10 ? "0" + endmes : endmes) +
    "-" +
    (enddia < 10 ? "0" + enddia : enddia);
$("#modalAgenda input[name='end']").val(endFormact);

// Preencher o campo endTime com 9h
let endTimeSelect = document.querySelector('select[name="endTime"]');
endTimeSelect.value = "09:00";

calendar.unselect();
```

Elaborado pela autora

9. Manipulação das datas de fim do evento selecionado:
    - *let end = new Date(evento.end):* Converte a data de fim do evento selecionado em um objeto de data.
    - *let endano = end.getFullYear():* Obtém o ano da data de fim.
    - *let endmes = end.getMonth() + 1:* Obtém o mês da data de fim (+1 porque os meses são indexados de 0 a 11).
    - *let enddia = end.getDate() – 1:* Obtém o dia do mês da data de fim e subtrai 1 para evitar conflito na hora de formatar.
    - *let endFormact = ...:* Cria uma string formatada da data de fim no formato “aaaa-mm-dd”.
    - *$(“#modalAgenda input[name='end']”).val(endFormact):* Define o valor do campo de data de fim do formulário no modal com a data formatada.
10. Preenchimento do campo de hora de fim:
    - Obtém o seletor do campo de hora de fim.
    - Define o valor do campo de hora de fim como “09:00”.
11. *calendar.unselect():* Cancela a seleção do evento no calendário.

A Figura 30 mostra a tela de agenda, que ocupa toda a proporção da tela e apresenta um calendário intuitivo para o agendamento de consultas odontológicas. O calendário possui três visualizações diferentes: mês, semana e lista. Cada dia do calendário exibe as consultas agendadas para esse dia.

Figura 30 - Tela de Agenda




Elaborado pela autora

### 3.3.4.3. Modal para agendamento

No arquivo “*blade.php*”, representado na Figura 31, é definido o modal de agendamento.

Figura 31 – Calendário – Modal Agendamento

```
<div class="modal fade bd-example-modal-lg" id="modalAgenda" tabindex="-1" role="dialog" aria-labelledby="modalTitle"
    aria-hidden="true" data-route-get-patients="{{ route('routeGetPatients', isset($id) ? ['id' => $id] : []) }}"
    data-route-get-dentists="{{ route('routeGetDentists', isset($id) ? ['id' => $id] : []) }}"
    data-route-get-clinicals="{{ route('routeGetClinicals', isset($id) ? ['id' => $id] : []) }}">
    <div class="modal-dialog modal-lg" role="document">
        <div class="modal-content">
            <div class="modal-header">
                <h5 class="modal-title" id="modalTitle">Agenda</h5>
                <button type="button" class="close" data-dismiss="modal" aria-label="Fechar">
                    <span aria-hidden="true">&times;</span>
                </button>
            </div>
            <div class="modal-body">

                <div id="message"></div>

                <form id="formAgenda" action="">

                    {!! csrf_field() !!}
```

Elaborado pela autora

O modal é criado como uma “*div*” com ID “*modalAgenda*”, ele possui um tamanho grande e é ativado ao clicar em um elemento específico, dentro do modal, há uma estrutura de conteúdo definida pela classe “*modal-content*”. O cabeçalho do modal contém um título “Agenda” com a classe “*modal-title*” e o ID “*modalTitle*”. Além disso, há um botão “Fechar” com a classe “close” e o atributo “*data-dismiss*” definido como “*modal*”, permitindo que o modal seja fechado ao clicar no botão.

No corpo do modal, há um elemento “*div*” com o ID “*message*”, esse elemento é utilizado para exibir mensagens de erro ou sucesso relacionadas ao agendamento. Dentro do *modal* também é definido um formulário com o ID “*formAgenda*”, o formulário possui um atributo “*action*” vazio, indicando que a rota para o envio do formulário será definida por meio do *JavaScript*. O formulário contém um campo para proteção contra ataques *CSRF* (*Cross-Site Request Forgery*) definido por *“{!! csrf_field() !!}”,* esse campo é importante para garantir a segurança do envio de dados no Laravel.

O modal também faz uso de atributos “*data-route-get-patients*”, “*data-route-get-dentists*” e “*data-route-get-clinicals*”, que armazenam as rotas para obter informações sobre pacientes, dentistas e clínicas, respectivamente. Essas rotas são utilizadas para carregar as opções nos campos do formulário de agendamento.

As Figuras 32, 33 e 34 apresentam a criação dos campos do modal de agendamento. Esses campos são parte do formulário que permite ao usuário inserir informações relacionadas ao agendamento.

Figura 32 – Calendário – Modal Agendamento

```
<div class="row">
    <div class="form-group col-md-6">
        <label>{{ _('Data Inicio') }} <span class="campo-obrigatorio"
                style="color: grey; margin-left: 5px;">*</span></label>
        <div class="row">
            <div class="col-md-6">
                <input type="date" name="start" class="form-control" required>
            </div>
            <div class="col-md-6">
                <select name="startTime" class="form-control" required>
                    <!-- Opções de horários serão preenchidas dinamicamente -->
                </select>
            </div>
        </div>
    </div>
    <div class="form-group col-md-6">
        <label>{{ _('Data Fim') }}<span class="campo-obrigatorio" style="color: grey; margin-left: 5px;">*</span></label>
        <div class="row">
            <div class="col-md-6">
                <input type="date" name="end" class="form-control" required disabled>
            </div>
            <div class="col-md-6">
                <select name="endTime" class="form-control" required disabled>
                    <!-- Opções de horários serão preenchidas dinamicamente -->
                </select>
            </div>
        </div>
    </div>
```

Elaborado pela autora

Na Figura 32, são definidos os campos relacionados à data e hora de início, e data e hora de término do agendamento. No primeiro grupo de campos, temos a “Data de Início” e a “Hora de Início”, a “Data de Início” é representada por um input do tipo “*date*” e a “Hora de Início” é representada por um “*select*”, onde o usuário pode escolher o horário de início do agendamento, ambos os campos são obrigatórios.

No segundo grupo de campos, temos a “Data de Término” e a “Hora de Término”, a “Data de Término” também é um *input* do tipo “*date*”, mas nesse caso, o campo está desabilitado (*disabled*), a “Hora de Término” é um “*select*”, semelhante ao campo de início, e está desabilitado. Essa desabilitação dos campos de término está relacionada à lógica de agendamento, onde o usuário pode selecionar a data e hora de início primeiro e, em seguida, o sistema calcula a data e hora de término com base no início e na duração do agendamento.

Figura 33 – Calendário – Modal Agendamento

```
<div class="row">
    <div class="form-group col-md-12">
        <label>{{ _('Paciente') }}<span class="campo-obrigatorio"
                style="color: grey; margin-left: 5px;">*</span></label>
        <select id="select-paciente" name="id_paciente" class="form-control" required>

            <option id="option-selecione-patient" value="">Selecione</option>

            @foreach ($patients as $patient)
                <option id="option-value-patient" value="{{ $patient->id }}">
                    {{ $patient->nome }}
                </option>
            @endforeach
        </select>
    </div>
</div>
```

Elaborado pela autora

Na Figura 33, temos a criação de um campo de seleção para escolher o paciente no formulário de agendamento. O campo é representado por um “*select*” com o ID “*select-paciente*” e o nome “*id_paciente*”, indicando que o valor selecionado será enviado como “*id_paciente*” no formulário.

Dentro do “*select*”, há uma primeira opção vazia com o valor “ “, indicada como “Selecione”, essa opção serve como um *placeholder* para instruir o usuário a escolher uma opção válida. Abaixo dessa opção vazia, há um loop “*foreach*” que itera sobre a variável “$*patients*”, que contém uma lista de pacientes disponíveis para o agendamento, dentro do loop, para cada paciente, é criada uma opção com o valor igual ao ID do paciente e o texto sendo o nome do paciente, dessa forma, o usuário pode selecionar o paciente desejado a partir das opções listadas.

Figura 34 – Calendário – Modal Agendamento

```
<div class="row">
    <div class="form-group col-md-6">
        <label>{{ _('Dentista') }}<span class="campo-obrigatorio"
                style="color: grey; margin-left: 5px;">*</span></label>
        <select id="select-dentista" name="id_dentista" class="form-control" required>

            <option id="option-selecione-dentist" value="">Selecione
            </option>

            @foreach ($dentists as $dentist)
                <option id="option-value-dentist" value="{{ $dentist->id }}">
                    {{ $dentist->nome }}</option>
            @endforeach
        </select>
    </div>
    <div class="form-group col-md-6">
        <label>{{ _('Clínica') }}<span class="campo-obrigatorio"
                style="color: grey; margin-left: 5px;">*</span></label>
        <select id="select-clinica" name="id_clinica" class="form-control" required>

            <option id="option-selecione-clinical" value="">Selecione
            </option>

            @foreach ($clinicals as $clinical)
                <option id="option-value-clinical" value="{{ $clinical->id }}">
                    {{ $clinical->nome }} -
                    {{ $clinical->cidade }}</option>
            @endforeach
        </select>
    </div>
</div>
```

Elaborado pela autora

Na Figura 34, temos a criação de dois campos de seleção: um para escolher o dentista e outro para escolher a clínica no formulário de agendamento. Ambos os campos são representados por “*select*”, um com o ID “*select-dentista*” e nome “*id_dentista*”, e o outro com o ID “*select-clinica*” e nome “*id_clinica*”. Isso significa que os valores selecionados nesses campos serão enviados como “*id_dentista*” e “*id_clinica*” no formulário, respectivamente. Dentro dos “*select*”, há uma primeira opção vazia com o valor “ “, indicada como “Selecione”, essa opção serve como um *placeholder* para instruir o usuário a escolher uma opção válida.

Em seguida, há um *loop* “*foreach*” para cada um dos “*select*”, no primeiro *loop*, itera sobre a variável “$*dentists*”, que contém uma lista de dentistas disponíveis para o agendamento. Dentro do *loop*, para cada dentista, é criada uma opção com o valor

igual ao ID do dentista e o texto sendo o nome do dentista, no segundo *loop*, itera sobre a variável “$*clinicals*”, que contém uma lista de clínicas disponíveis para o agendamento. Dentro do *loop*, para cada clínica, é criada uma opção com o valor igual ao ID da clínica e o texto sendo o nome da clínica seguido pelo nome da cidade da clínica.

Na Figura 35, temos a criação dos botões que compõem o formulário de agendamento. Esses botões são essenciais para permitir ao usuário interagir com o formulário, realizar ações como salvar o agendamento, fechar o modal e, se aplicável, excluir o agendamento existente.

Figura 35 – Calendário – Modal Agendamento

```
{{-- Buttons --}}
<div class="text-center">
    <div class="btn-group btn-group-toggle">
        <div class="text-right">
            <button type="button" class="btn btn-sm btn-success saveEvent"
                id="agendaSalvar">{{ _('Salvar') }}</button>
        </div>
        <div class="text-right">
            <button type="button" class="btn btn-sm btn-danger deleteEvent"
                id="agendaExcluir">{{ _('Excluir') }}</button>
        </div>
        <div class="text-right">
            <button type="button" class="btn btn-sm btn-primary"
                data-dismiss="modal">{{ _('Voltar') }}</button>
        </div>
    </div>
</div>
```

Elaborado pela autora

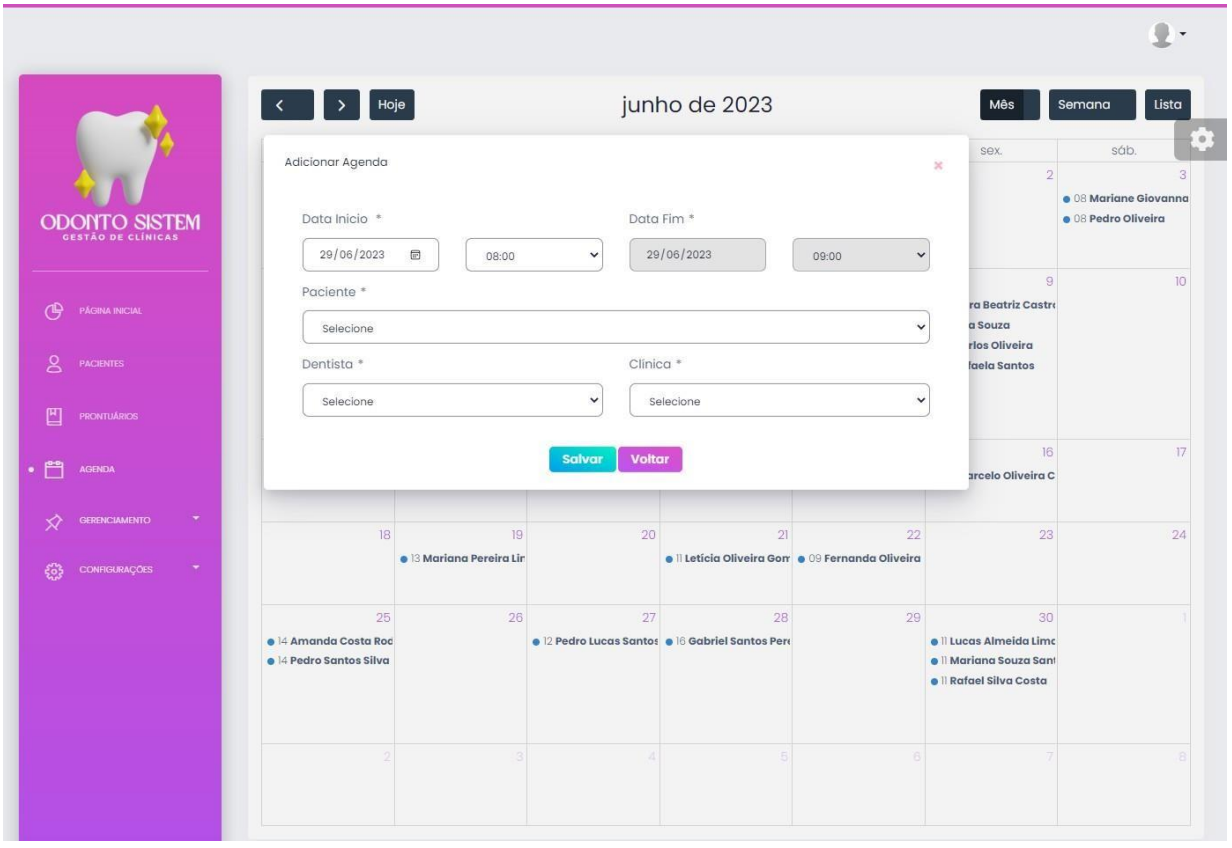
Os botões estão agrupados dentro de uma “<*div*>“ com a classe “*btn-group btn-group-toggle*”, que é uma classe do Bootstrap que cria um grupo de botões que podem ser alternados (*toggle*) entre si.

O primeiro botão é definido com o ID “agendaSalvar”, esse botão possui a classe “*btn btn-sm btn-success*”, indicando que é um botão pequeno e tem uma cor de fundo verde, que é associada a ações positivas, como salvar. O segundo botão é definido com o ID “agendaExcluir”, este botão possui a classe “*btn btn-sm btn-*

*danger*”, o que indica que é um botão pequeno, com uma cor de fundo vermelha, que é associada a ações de exclusão.

O terceiro botão possui a classe “*btn btn-sm btn-primary*”, indicando que é um botão pequeno, com uma cor de fundo azul, que é associada a ações de voltar ou cancelar. Ele também tem o atributo *data-dismiss=“modal*”, que está relacionado ao modal onde o formulário está contido, isso significa que, ao clicar neste botão, o modal será fechado, ocultando o formulário de agendamento. A Figura 36 exibe o modal utilizado para realizar agendamentos no calendário.

Figura 36 - Modal Agendamento




Elaborado pela autora

### 3.3.4.4. Cadastrar ou Editar um evento

No código *JavaScript*, as ações de salvar e excluir eventos no calendário são realizadas por meio de funções, quando o botão “*saveEvent*” é clicado, o *JavaScript* recupera os dados do formulário de agendamento, como a data de início, a hora de

início, a data de término, a hora de término, o ID do paciente, o ID do dentista e o ID da clínica. Essas informações são usadas para criar os eventos no formato apropriado e realizar as validações necessárias.

As validações são feitas para garantir que os horários de início e término do evento estejam dentro de um intervalo aceitável, entre 08:00 e 18:00 horas. Se os horários estiverem fora desse intervalo ou se o horário final for anterior ao horário inicial, uma mensagem de erro é exibida ao usuário, informando a condição inválida. Além disso, é verificado se os campos obrigatórios do formulário foram preenchidos corretamente. Se o campo “Paciente”, “Dentista” ou “Clínica” estiver vazio, uma mensagem de erro é exibida ao usuário, informando que esses campos são obrigatórios.

Após as validações, é realizada uma verificação para detectar se existem eventos duplicados no calendário. Essa verificação é feita por meio da função “*checkDuplicateEvent*”, cujo trecho de código é mostrado na Figura 37.

Figura 37 – Calendário – Agendamento JavaScript

```
function checkDuplicateEvent(start = "", end = "", id_dentista = "") {
    const url =
        document.getElementById("agenda").dataset["routeCheckDuplicateEvent"];
    console.log(url);
    return new Promise((resolve, reject) => {
        $.ajax({
            url: url + "/" + start + "/" + end + "/" + id_dentista,
            type: "GET",
            success: function (response) {
                resolve(response.exists); // Resolve a promise com o resultado da verificação
            },
            error: function (xhr, status, error) {
                reject(error); // Rejeita a promise com o erro
            },
        });
    });
}
```

Elaborado pela autora

A função “*checkDuplicateEvent*” é responsável por verificar a existência de eventos duplicados no calendário, com base na data de início, data de término e ID do dentista informados como parâmetros. Essa função é utilizada para evitar que

seja criado um evento em um horário já ocupado por outro agendamento do mesmo dentista.

A função inicia recuperando a *URL* da rota a ser utilizada para fazer a verificação de duplicidade de eventos, essa é obtida a partir do atributo "*data-routeCheckDuplicateEvent*" do elemento com o ID "agenda" (calendário). Esse atributo contém a rota a ser acessada no servidor para executar a verificação. Em seguida, a função retorna uma nova *Promise*, que permite realizar a verificação de forma assíncrona. Dentro da *Promise*, é feita uma requisição *AJAX* do tipo *GET* para a *URL* definida, enviando como parâmetros a data de início, a data de término e o ID do dentista.

Quando a requisição é bem-sucedida (*success*), a *Promise* é resolvida com o resultado da verificação, representado pela propriedade "*exists*" do objeto "*response*", essa propriedade indica se existem eventos duplicados para o dentista no intervalo de tempo especificado. Caso existam eventos duplicados, o valor de "*exists*" será verdadeiro, caso contrário, será falso. Em caso de erro na requisição (*error*), a *Promise* é rejeitada com a mensagem de erro retornada pelo servidor. Essa função é útil para garantir que não haja agendamentos conflitantes para o mesmo dentista, evitando sobreposições de horários e assegurando que os horários de atendimento estejam disponíveis e precisos no calendário.

Após a validação de eventos duplicados realizada pela função "*checkDuplicateEvent*", caso não ocorram erros, o salvamento do agendamento continua. O trecho de código na Figura 38 apresenta essa parte do processo.

Figura 38 – Calendário – Agendamento JavaScript

```
// Evento não duplicado, prosseguir com o salvamento
getName("routeGetPatients", id_paciente)
    .then((nome) => {
        let title = nome;

        let Event = {
            title: title,
            start: start,
            end: end,
            id_paciente: id_paciente,
            id_dentista: id_dentista,
            id_clinica: id_clinica,
        };

        let route;

        if (id == "") {
            route = routeEvents("routeEventStore");
        } else {
            route = routeEvents("routeEventUpdate");
            Event.id = id;
            Event._method = "PUT";
        }

        sendEvent(route, Event, error);
    })
    .catch((error) => {
        console.log("Erro ao buscar o paciente: " + error);
    });
```

Elaborado pela autora

O trecho de código apresentado é parte desse processo, e a seguir, vou explicá-lo em detalhes.

1. *getName(“routeGetPatients”, id_paciente)*: A função “*getName*” é responsável por obter o nome do paciente a partir do ID do paciente (*id_paciente*), essa função é chamada passando a rota “*routeGetPatients*” e o ID do paciente como parâmetros. A função “*getName*” faz uma requisição *AJAX* para obter o nome do paciente e retorna uma *Promise* com o nome obtido.
2. *let Event = { ... }:* É criado um objeto Event que representa os dados do evento a ser adicionado no calendário. Esse objeto possui as seguintes propriedades:

- *title*: O título do evento, que é o nome do paciente.
- *start*: A data e hora de início do evento, obtidos anteriormente.
- *end*: A data e hora de término do evento, obtidos anteriormente.
- *id_paciente*: O ID do paciente selecionado.
- *id_dentista*: O ID do dentista selecionado.
- *id_clinica*: O ID da clínica selecionada.

3. *let route*: É criada a variável *route*, que será usada para armazenar a rota a ser utilizada para enviar o evento ao servidor.
4. *if (id == “”) { ... } else { ... }:* É feita uma verificação para determinar se o ID do evento está vazio ou não. Se estiver vazio, significa que é uma operação de criação de novo evento e, portanto, a rota utilizada será a de criação (*routeEventStore*), caso contrário, é uma operação de atualização de evento existente, e a rota utilizada será a de atualização (*routeEventUpdate*). Além disso, caso seja uma atualização, o ID do evento é adicionado ao objeto Event, bem como a propriedade “*_method*” com valor “*PUT*”, indicando que será uma requisição de atualização.
5. *sendEvent(route, Event, error):* É chamada a função “*sendEvent*” passando a rota, o objeto Event e o objeto “*error*” (que contém possíveis mensagens de erro). Essa função é responsável por enviar o evento ao servidor para salvamento ou atualização.
6. *.catch((error) => { ... }):* O método “.*catch*” é usado para tratar o caso de erro na *Promise* retornada pela função “*getName*”. Caso ocorra algum erro na busca do nome do paciente, uma mensagem de erro é registrada no console. Isso serve para fins de depuração, caso haja algum problema na obtenção dos dados do paciente.

Em resumo, essa parte do código trata a criação do evento a ser adicionado no calendário após a verificação de que não há eventos duplicados para o dentista no horário selecionado, o nome do paciente é obtido a partir do ID do paciente selecionado, e um objeto *Event* é criado com as informações do evento. Dependendo do caso (criação ou atualização), a rota adequada é definida, e então o evento é enviado ao servidor para ser salvo ou atualizado, caso ocorra algum erro durante o processo, uma mensagem é exibida no console para depuração.

#### 3.3.4.5. Exclusão de um evento

Quando o botão “*deleteEvent*” é clicado, é executado o trecho de código representado pela Figura 39.

Figura 39 – Calendário – Exclusão JavaScript

```
$(".deleteEvent").click(function () {
    let Event = {
        id: id,
        _method: "DELETE",
    };

    sendEvent(routeEvents("routeEventDelete"), Event);
});
```

Elaborado pela autora

Ao clicar nesse botão, a seguinte ação é realizada:

1. É criado um objeto *Event* que contém o ID do evento a ser deletado e a propriedade “*_method*” com o valor “*DELETE*”. O “*_method*” é um campo utilizado para realizar uma requisição *HTTP DELETE* quando o formulário está enviando um método *POST*, isso é necessário porque alguns navegadores não suportam a requisição *HTTP DELETE* diretamente em formulários *HTML*.
2. Em seguida, a função “*sendEvent*” é chamada com os parâmetros *routeEvents(“routeEventDelete*”) e o objeto *Event*. A função “*routeEvents*” é utilizada para obter a rota específica para a ação de deletar um evento (definida anteriormente nas rotas). A função “*sendEvent*” é responsável por enviar a requisição *HTTP DELETE* para o servidor, passando a rota e o objeto *Event*.

Essa sequência de ações resulta em uma requisição para o servidor para deletar o evento selecionado no calendário. A partir daí, a implementação do servidor irá processar a requisição e efetuar a exclusão do evento no banco de dados.

### 3.3.4.6. Event Click

O trecho de código das Figuras 40 e 41 representa o que ocorre quando um evento é clicado no calendário.

Figura 40 – Calendário – Evento de Click

```
eventClick: function (evento) {
    clearMessages("#message");
    resetForm("#formAgenda");
    $("#modalAgenda").modal("show");
    $("#modalAgenda #modalTitle").text("Alterar Agenda");
    $("#modalAgenda button.deleteEvent").css("display", "flex");

    console.log(evento);
    // ID
    id = evento.event.id;

    // Data Start
    let start = new Date(evento.event.start);
    let startDate = start.toISOString().slice(0, 10);
    let startTime = start.toTimeString().slice(0, 5);
    $("#modalAgenda input[name='start']").val(startDate);
    $("#modalAgenda input[name='startTime']").val(startTime);

    // Preencher o seletor de hora de início
    let startTimeSelect = document.querySelector(
        'select[name="startTime"]'
    );
    let [startHour, startMinute] = startTime.split(":");
    startTimeSelect.value = `${startHour.padStart(
        2,
        "0"
    )}:${startMinute.padStart(2, "0")}`;

    // Data End
    let end = new Date(evento.event.end);
    let endDate = end.toISOString().slice(0, 10);
    let endTime = end.toTimeString().slice(0, 5);
    $("#modalAgenda input[name='end']").val(endDate);
    $("#modalAgenda input[name='endTime']").val(endTime);
```

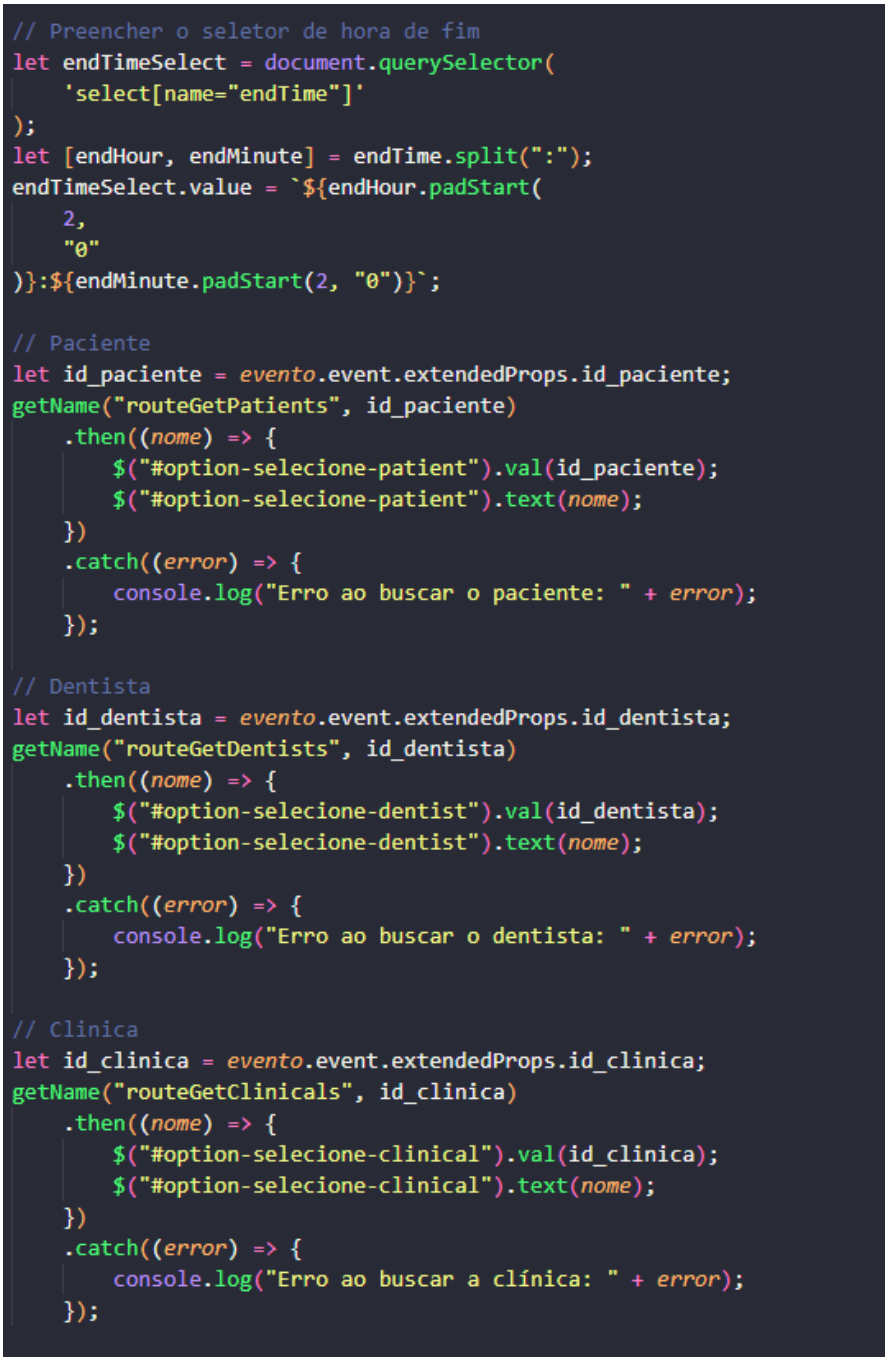
Elaborado pela autora

A função “*eventClick*” é chamada com o evento como parâmetro, e dentro dela são realizadas várias ações para apresentar os detalhes do evento no *modal* de agendamento.

Aqui está o passo a passo do que acontece quando o evento é clicado:

1. Limpeza e preparação do modal:
    - A função *clearMessages(“#message”)* é chamada para limpar quaisquer mensagens de erro ou aviso que possam estar visíveis no *modal*.
    - A função *resetForm(“#formAgenda”)* é chamada para redefinir o formulário dentro do modal, deixando-o em seu estado inicial.
    - O *modal* de agendamento é mostrado na tela com a chamada *$(“#modalAgenda”).modal(“show”).*
    - O título do modal é definido como “Alterar Agenda” usando *$(“#modalAgenda #modalTitle”).text(“Alterar Agenda”).*
    - O botão de excluir evento (*button.deleteEvent*) é exibido, pois o usuário deseja editar o evento existente.

Figure 41 – Calendário – Evento de Click



```
// Preencher o seletor de hora de fim
let endTimeSelect = document.querySelector(
    'select[name="endTime"]'
);
let [endHour, endMinute] = endTime.split(":");
endTimeSelect.value = `${endHour.padStart(
    2,
    "0"
)}:${endMinute.padStart(2, "0")}`;

// Paciente
let id_paciente = evento.event.extendedProps.id_paciente;
getName("routeGetPatients", id_paciente)
    .then((nome) => {
        $("#option-selecione-patient").val(id_paciente);
        $("#option-selecione-patient").text(nome);
    })
    .catch((error) => {
        console.log("Erro ao buscar o paciente: " + error);
    });

// Dentista
let id_dentista = evento.event.extendedProps.id_dentista;
getName("routeGetDentists", id_dentista)
    .then((nome) => {
        $("#option-selecione-dentist").val(id_dentista);
        $("#option-selecione-dentist").text(nome);
    })
    .catch((error) => {
        console.log("Erro ao buscar o dentista: " + error);
    });

// Clinica
let id_clinica = evento.event.extendedProps.id_clinica;
getName("routeGetClinicals", id_clinica)
    .then((nome) => {
        $("#option-selecione-clinical").val(id_clinica);
        $("#option-selecione-clinical").text(nome);
    })
    .catch((error) => {
        console.log("Erro ao buscar a clínica: " + error);
    });
```

Elaborado pela autora

2. Obtenção dos detalhes do evento:
    - O ID do evento é obtido a partir do objeto “*evento.event.id*” e armazenado em uma variável global ID. Esse ID será usado posteriormente para identificar o evento a ser editado ou excluído.

- As datas e horários de início e fim do evento são obtidos a partir do objeto "*evento.event.start*" e "*evento.event.end*", respectivamente. Essas informações são formatadas adequadamente e exibidas nos campos correspondentes do modal.
- O nome do paciente, do dentista e da clínica associados ao evento também são obtidos a partir das propriedades "*id_paciente*", "*id_dentista*" e *"id_clinica*" no objeto "*evento.event.extendedProps*". Esses valores são buscados por meio das funções "*getName("routeGetPatients", id_paciente*)", "*getName("routeGetDentists", id_dentista*)" e "*getName("routeGetClinicals", id_clinica*)" respectivamente, que fazem uma requisição para obter os nomes dos pacientes, dentistas e clínicas correspondentes com base em seus IDs. Em seguida, os nomes são exibidos nos campos de seleção adequados do modal.

Assim, quando o usuário clica em um evento no calendário, o modal de agendamento é preenchido com as informações do evento, permitindo que o usuário visualize e edite os detalhes do evento de forma interativa.

#### 3.3.4.7. Event Drop e Resize

Os eventos "*eventDrop*" e "*eventResize*" são responsáveis por permitir que o usuário arraste e redimensione os eventos no calendário, respectivamente. Ambos são atribuídos da mesma forma e são representados pelas Figuras 42 e 43.

Figura 42 – Calendário – Evento de Drop e Resize

```
// Data Start
let start = new Date(event.event.start);
let startano = start.getFullYear();
let startmes = start.getMonth() + 1;
let startdia = start.getDate();
let starthora = start.getHours();
let startminuto = start.getMinutes();
let startsegundo = start.getSeconds();
let starthoraFormatada = starthora.toString().padStart(2, "0");
let startminutoFormatado = startminuto.toString().padStart(2, "0");
let startsegundoFormatado = startsegundo
    .toString()
    .padStart(2, "0");
let startFormact =
    startano +
    "-" +
    (startmes < 10 ? "0" + startmes : startmes) +
    "-" +
    (startdia < 10 ? "0" + startdia : startdia) +
    " " +
    starthoraFormatada +
    ":" +
    startminutoFormatado +
    ":" +
    startsegundoFormatado;
```

Elaborado pela autora

O trecho de código fornecido é responsável por formatar a data e hora de início de um evento no calendário. Ele utiliza o objeto “*Date*” do *JavaScript* para extrair informações como ano, mês, dia, hora, minuto e segundo da data de início do evento, em seguida, essas informações são formatadas em uma string no padrão “AAAA-MM-DD HH:mm:ss. A formatação é realizada usando métodos como *“getFullYear()”, “getMonth()”, “getDate()”, “getHours()”, “getMinutes()”, “getSeconds()”* e *“padStart()”.*

Figura 43 – Calendário – Evento de Drop e Resize

```
    // Data End
    let end = new Date(event.event.end);
    let endano = end.getFullYear();
    let endmes = end.getMonth() + 1;
    let enddia = end.getDate();
    let endhora = end.getHours();
    let endminuto = end.getMinutes();
    let endsegundo = end.getSeconds();
    let endhoraFormatada = endhora.toString().padEnd(2, "0");
    let endminutoFormatado = endminuto.toString().padEnd(2, "0");
    let endsegundoFormatado = endsegundo.toString().padEnd(2, "0");
    let endFormact =
        endano +
        "-" +
        (endmes < 10 ? "0" + endmes : endmes) +
        "-" +
        (enddia < 10 ? "0" + enddia : enddia) +
        " " +
        endhoraFormatada +
        ":" +
        endminutoFormatado +
        ":" +
        endsegundoFormatado;

    let newEvent = {
        _method: "PUT",
        id: event.event.id,
        start: startFormact,
        end: endFormact,
    };

    sendEvent(routeEvents("routeEventUpdate"), newEvent);
},

events: routeEvents("routeLoadEvents"),
```

Elaborado pela autora

O código apresentado é responsável por formatar a data e hora de término de um evento no calendário, assim como o trecho anterior formatou a data e hora de início. Esse código utiliza o objeto “*Date*” do *JavaScript* para extrair informações como ano, mês, dia, hora, minuto e segundo da data de término do evento e, em seguida, essas informações são formatadas em uma string no padrão “AAAA-MM-DD HH:mm:ss. A formatação é realizada usando métodos como “*getFullYear()”, “getMonth()”, “getDate()”, “getHours()”, “getMinutes()”, “getSeconds()”* e *“padEnd()”.*

Essa string formatada é então utilizada para atualizar o evento no calendário, sendo enviada ao *back-end* por meio de uma requisição *PUT* com os novos valores de data e hora de início e término. A atualização do evento ocorre quando o usuário arrasta o evento para uma nova data ou hora no calendário, e essa nova data e hora são refletidas no calendário, bem como no armazenamento dos eventos no *back-end.*

## 3.4. Validações

As validações desempenham um papel fundamental no funcionamento do sistema, garantindo a integridade dos dados e a consistência das informações inseridas pelos usuários. Elas asseguram que apenas dados válidos sejam aceitos e processados, evitando erros, inconsistências e potenciais problemas no sistema. No contexto desse sistema, várias validações são aplicadas para garantir que os dados fornecidos pelos usuários estejam corretos e atendam aos critérios estabelecidos.

A abordagem de utilizar uma "* *" com o ID "*message*" para exibir mensagens de validação é uma forma eficiente e prática de fornecer *feedback* ao usuário durante o preenchimento de formulários ou interações no sistema. Com o auxílio do *JavaScript*, é possível implementar validações personalizadas e exibir mensagens adequadas conforme o contexto, tornando a experiência do usuário mais clara e amigável.

Ao realizar as validações no *JavaScript*, é possível verificar os dados inseridos pelo usuário em tempo real, antes de enviar qualquer informação para o servidor. Dessa forma, as mensagens de *feedback* podem ser exibidas instantaneamente, auxiliando os usuários a compreenderem quais campos precisam ser corrigidos ou preenchidos corretamente.

A personalização das mensagens de validação é fundamental para fornecer orientações claras e específicas aos usuários. Por exemplo, se um campo obrigatório for deixado em branco, uma mensagem como "Este campo é obrigatório" pode ser exibida. Da mesma forma, para validações específicas, como formato de data ou *CPF* inválido, mensagens mais detalhadas podem ser mostradas, orientando o usuário sobre como solucionar o problema.

Utilizando esse método de validação e exibição de mensagens, é possível melhorar a usabilidade do sistema, reduzindo erros e frustrações dos usuários, além disso, garante-se a integridade dos dados enviados para o servidor. Essa abordagem flexível permite que a equipe de desenvolvimento implemente validações específicas para cada campo do formulário, adaptando-as conforme as necessidades do sistema e das interações do usuário.

### 3.4.1. Datas

Na Figura 44 e 45, apresenta-se um trecho de código referente à tela de pacientes, onde ocorrem as validações para o campo “Data da primeira consulta”.

Figura 44 – Validação Data Primeira Consulta

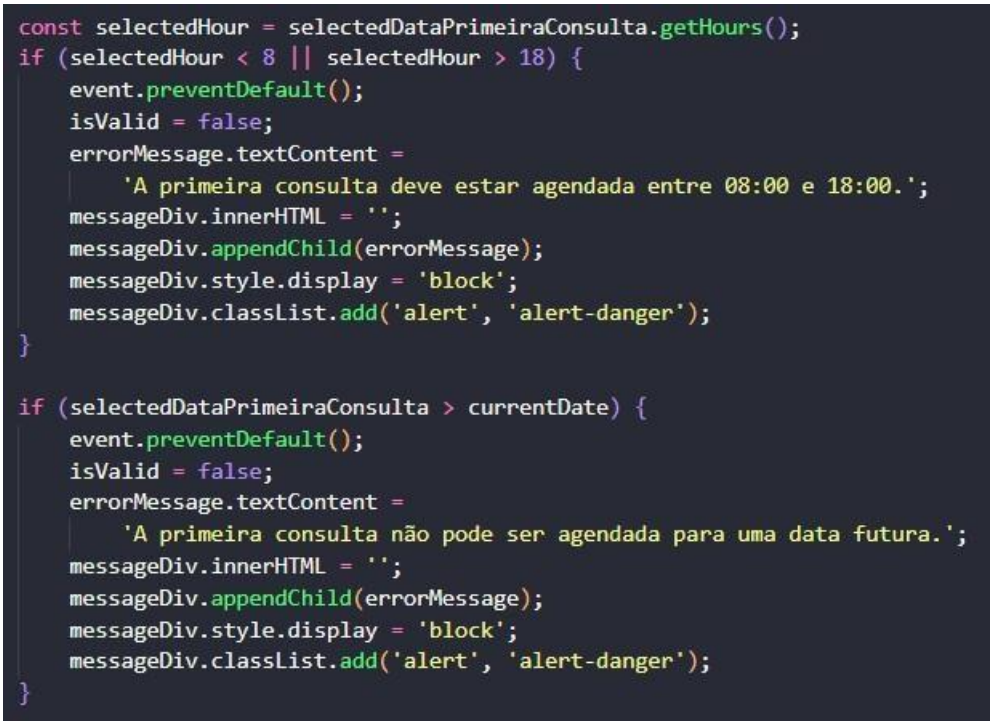
```
if (selectedDataNascimento > selectedDataPrimeiraConsulta) {
    event.preventDefault();
    isValid = false;
    errorMessage.textContent =
        'A primeira consulta não pode ser agendada antes da data de nascimento.';
    messageDiv.innerHTML = '';
    messageDiv.appendChild(errorMessage);
    messageDiv.style.display = 'block';
    messageDiv.classList.add('alert', 'alert-danger');
}

const sixMonthsAfterBirth = new Date(selectedDataNascimento);
sixMonthsAfterBirth.setMonth(sixMonthsAfterBirth.getMonth() + 6);

const diffInMonths = (selectedDataPrimeiraConsulta.getFullYear() - selectedDataNascimento
        .getFullYear()) * 12 +
    (selectedDataPrimeiraConsulta.getMonth() - selectedDataNascimento.getMonth());

if (diffInMonths < 6) {
    event.preventDefault();
    isValid = false;
    errorMessage.textContent =
        'A primeira consulta deve ser agendada no mínimo 6 meses após a data de nascimento.';
    showMessage(errorMessage);
}
```

Elaborado pela autora

A primeira validação compara a data de nascimento do paciente com a data da primeira consulta, se a data da primeira consulta for anterior à data de nascimento, uma mensagem de erro é exibida, alertando que a consulta não pode ser agendada antes do nascimento do paciente.

Em seguida, é calculado o período de seis meses após a data de nascimento do paciente, usando o objeto “*Date*”. O código determina a diferença em meses entre a data da primeira consulta e a data de nascimento, e se esse valor for menor que seis, outra mensagem de erro é mostrada, informando que a primeira consulta deve ser agendada com pelo menos seis meses de diferença após o nascimento do paciente.

Figura 45 – Validação Data Primeira Consulta



```
const selectedHour = selectedDataPrimeiraConsulta.getHours();
if (selectedHour < 8 || selectedHour > 18) {
    event.preventDefault();
    isValid = false;
    errorMessage.textContent =
        'A primeira consulta deve estar agendada entre 08:00 e 18:00.';
    messageDiv.innerHTML = '';
    messageDiv.appendChild(errorMessage);
    messageDiv.style.display = 'block';
    messageDiv.classList.add('alert', 'alert-danger');
}

if (selectedDataPrimeiraConsulta > currentDate) {
    event.preventDefault();
    isValid = false;
    errorMessage.textContent =
        'A primeira consulta não pode ser agendada para uma data futura.';
    messageDiv.innerHTML = '';
    messageDiv.appendChild(errorMessage);
    messageDiv.style.display = 'block';
    messageDiv.classList.add('alert', 'alert-danger');
}
```

Elaborado pela autora

A primeira validação compara a hora selecionada para a primeira consulta (*selectedHour*) e verifica se ela está fora do intervalo permitido entre 08:00 e 18:00. Se a hora estiver fora desse intervalo, uma mensagem de erro é exibida, informando que a primeira consulta deve ser agendada apenas entre esses horários válidos.

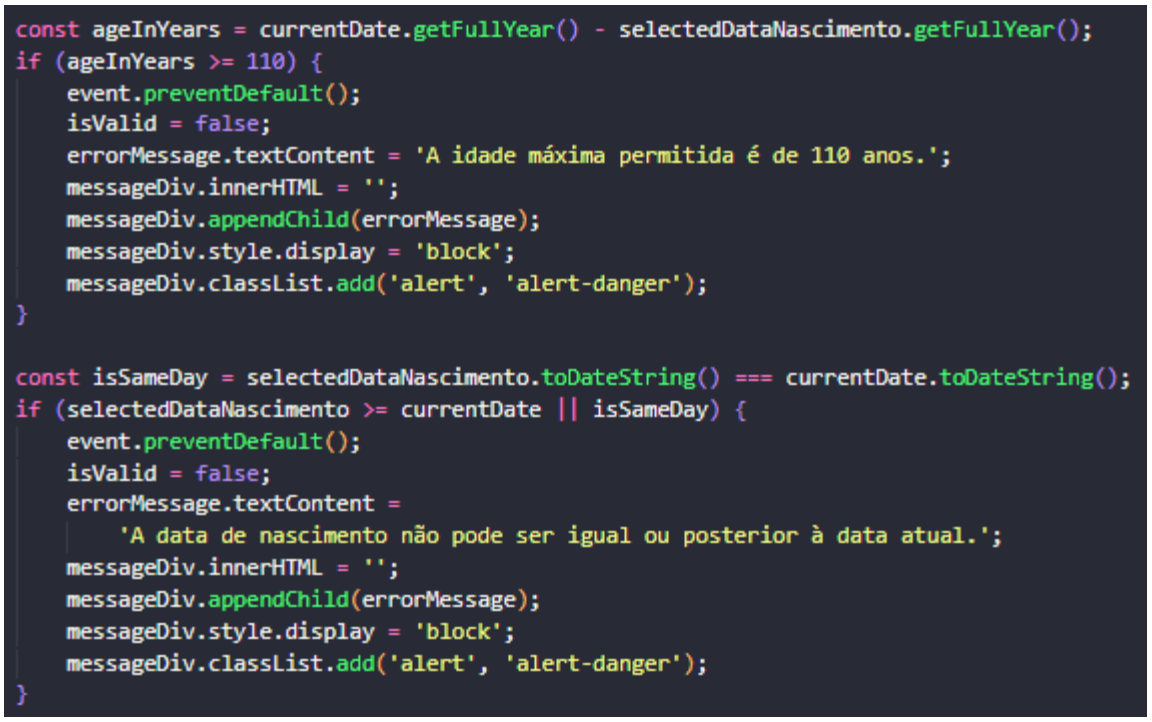
A abordagem de definir um intervalo de horário permitido para as consultas, como entre 08:00 e 18:00, é uma prática importante e consistente que pode ser aplicada em várias telas do sistema que envolvem agendamento de horários. Ao adotar essa abordagem em diferentes partes do sistema, como no prontuário e

agenda, garante-se a coerência e padronização das regras de agendamento em toda a aplicação.

A segunda validação verifica se a data da primeira consulta está no futuro em relação à data atual, se a consulta estiver marcada para uma data futura, uma mensagem de erro é mostrada, alertando que a primeira consulta não pode ser agendada para datas que ainda não ocorreram.

O código da Figura 46 apresenta abordagens responsáveis e cuidadosas para lidar com informações relacionadas à idade dos pacientes.

Figura 46 – Validação Data Nascimento



```
const ageInYears = currentDate.getFullYear() - selectedDataNascimento.getFullYear();
if (ageInYears >= 110) {
    event.preventDefault();
    isValid = false;
    errorMessage.textContent = 'A idade máxima permitida é de 110 anos.';
    messageDiv.innerHTML = '';
    messageDiv.appendChild(errorMessage);
    messageDiv.style.display = 'block';
    messageDiv.classList.add('alert', 'alert-danger');
}

const isSameDay = selectedDataNascimento.toDateString() === currentDate.toDateString();
if (selectedDataNascimento >= currentDate || isSameDay) {
    event.preventDefault();
    isValid = false;
    errorMessage.textContent =
        'A data de nascimento não pode ser igual ou posterior à data atual.';
    messageDiv.innerHTML = '';
    messageDiv.appendChild(errorMessage);
    messageDiv.style.display = 'block';
    messageDiv.classList.add('alert', 'alert-danger');
}
```

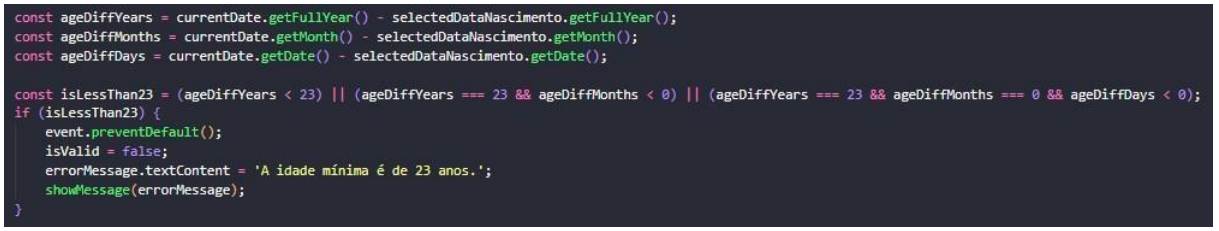
Elaborado pela autora

A primeira validação examina a idade do paciente, restringindo-a a um limite máximo de 110 anos, essa regra de negócio é importante para evitar o cadastro de pacientes com idades inválidas ou incoerentes, garantindo que o sistema funcione adequadamente e que os dados reflitam a realidade. Limitar a idade máxima é uma prática comum em sistemas médicos, pois raramente se espera que pacientes ultrapassem esse limite.

A segunda validação lida com a data de nascimento do paciente, ela verifica se a data fornecida não é igual ou posterior à data atual. Essa medida evita o cadastro de pacientes com datas de nascimento futuras ou datas incorretas, o que poderia comprometer a precisão das informações e a eficiência do sistema.

A Figura 47 representa uma validação adicional na tela de cadastro de dentistas, na qual é essencial verificar a idade do profissional antes de permitir o registro. Essa validação se assemelha à que foi abordada anteriormente para os pacientes, garantindo a integridade e coerência dos dados relacionados ao cadastro dos profissionais de odontologia.

Figura 47 – Validação Idade Dentista



```
const ageDiffYears = currentDate.getFullYear() - selectedDataNascimento.getFullYear();
const ageDiffMonths = currentDate.getMonth() - selectedDataNascimento.getMonth();
const ageDiffDays = currentDate.getDate() - selectedDataNascimento.getDate();

const isLessThan23 = (ageDiffYears < 23) || (ageDiffYears === 23 && ageDiffMonths < 0) || (ageDiffYears === 23 && ageDiffMonths === 0 && ageDiffDays < 0);
if (isLessThan23) {
    event.preventDefault();
    isValid = false;
    errorMessage.textContent = 'A idade mínima é de 23 anos.';
    showMessage(errorMessage);
}
```

Elaborado pela autora

Nesse caso, o sistema verifica se o profissional possui uma idade mínima de 23 anos antes de permitir o registro, a verificação é feita com base na data de nascimento fornecida pelo usuário e a data atual. Para calcular a diferença de idade, o código utiliza a data atual do sistema e a data de nascimento selecionada. É realizada uma comparação entre os anos, meses e dias para determinar se a idade é menor que 23 anos. Essa abordagem é bastante precisa e leva em consideração até mesmo a diferença de dias para garantir que o dentista tenha completado 23 anos de idade.

Caso a idade do dentista seja menor que 23 anos, a validação é acionada e a operação de registro é impedida, o sistema exibe uma mensagem de erro personalizada, informando que a idade mínima permitida é de 23 anos.

### 3.4.2. CEP, CPF e CRO

O trecho de código apresentado na Figura 48 refere-se a validações de *CPF* e *CEP* na tela de cadastro de pacientes, também são utilizadas em outras telas do sistema.

Figura 48 – Validação CEP e CPF

```
const cepInput = document.querySelector('input[name="cep"]');
const cpfInput = document.querySelector('input[name="cpf"]');
const cepValue = cepInput.value.trim();
const cpfValue = cpfInput.value.trim();

if (!/^\d{8}$/.test(cepValue)) {
    event.preventDefault();
    isValid = false;
    errorMessage.textContent = 'O campo de CEP deve conter exatamente 8 dígitos numéricos.';
    showMessage(errorMessage);
}

if (!/^\d{11}$/.test(cpfValue)) {
    event.preventDefault();
    isValid = false;
    errorMessage.textContent =
        'O campo de CPF deve conter exatamente 11 dígitos numéricos.';
    showMessage(errorMessage);
}
```

Elaborado pela autora

Para a validação do *CEP*, o código utiliza uma expressão regular que verifica se o valor contém exatamente 8 dígitos numéricos, caso o *CEP* não atenda a essa condição, é exibida uma mensagem de erro informando que o campo deve conter exatamente 8 dígitos.

Já para a validação do *CPF*, outra expressão regular é utilizada para verificar se o valor contém exatamente 11 dígitos numéricos, se o *CPF* não estiver de acordo com esse formato, uma mensagem de erro é exibida, indicando que o campo deve conter exatamente 11 dígitos.

Essas validações são importantes para garantir que os campos de *CEP* e *CPF* estejam preenchidos corretamente antes de serem cadastrados no sistema. Ao realizar essas verificações no momento do cadastro, é possível evitar a inclusão de

informações inválidas ou incompletas, garantindo a qualidade e confiabilidade dos dados armazenados.

O trecho de código da Figura 49 apresenta a validação para o campo de *CRO* (Cadastro Regional de Odontologia) na tela de cadastro de dentista. Essa validação é essencial para garantir que o valor fornecido pelo usuário esteja em conformidade com o formato esperado para o número do *CRO*.

Figura 49 – Validação CRO

```
// Verifique se o CRO contém apenas dígitos e possui o número correto de dígitos (6).
if (!/^\d{6}$/.test(croValue)) {
    event.preventDefault();
    isValid = false;
    errorMessage.textContent = 'O CRO deve conter exatamente 6 dígitos numéricos.';
    showMessage(errorMessage);
}
```

Elaborado pela autora

A expressão regular utilizada no código verifica se o valor do *CRO* contém exatamente 6 dígitos numéricos, caso o *CRO* não atenda a essa condição, uma mensagem de erro é exibida, informando que o campo deve conter exatamente 6 dígitos. Essa validação é de suma importância, pois o *CRO* é um número único e específico para cada dentista, emitido pelo Conselho Regional de Odontologia. Garantir que o *CRO* seja inserido corretamente é fundamental para manter a integridade e precisão dos registros de profissionais cadastrados no sistema.

Ao aplicar essa validação no campo de *CRO*, evita-se o cadastro de informações inválidas ou incorretas, mantendo a qualidade dos dados e a confiabilidade das informações sobre os dentistas registrados na plataforma. Além disso, a validação do *CRO* também contribui para a padronização dos registros no sistema, facilitando a busca e identificação de dentistas por meio do número de *CRO*. Isso proporciona uma melhor organização e gestão dos profissionais cadastrados, além de garantir a conformidade com as normas e regulamentações do órgão responsável pela regulamentação da profissão.

### 3.4.3. Datas do Prontuário

Essa parte do código da Figura 50 representa mais validações para a data e hora da consulta no prontuário médico, com foco em garantir que as marcações estejam dentro de um intervalo adequado e evitando agendamentos fora do horário de trabalho e do período atual.

Figura 50 – Validação Data Prontuário

```
// Validar horário entre 08:00 e 18:00
const hour = selectedData.getHours();
const isOutsideWorkingHours = hour < 8 || hour >= 18;
if (isOutsideWorkingHours) {
    event.preventDefault();
    isValid = false;
    errorMessage.textContent = 'O horário deve estar entre 08:00 e 18:00.';
    showMessage(errorMessage);
}

// Verificar se a data e hora são maiores que a atual
if (selectedData > currentDate) {
    event.preventDefault();
    isValid = false;
    errorMessage.textContent = 'A data e hora não podem ser maiores que a atual.';
    showMessage(errorMessage);
}
```

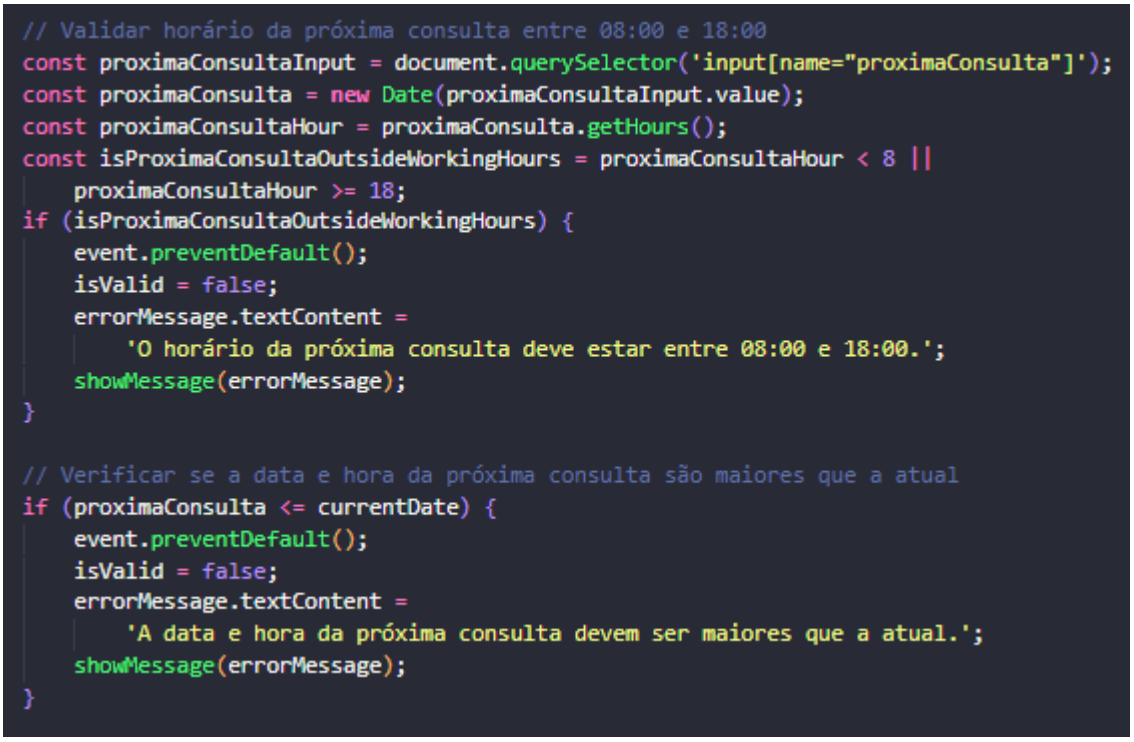
Elaborado pela autora

A primeira validação verifica se o horário selecionado para a consulta está fora do horário de trabalho, que é entre 08:00 e 18:00, caso a consulta seja agendada fora desse intervalo, o sistema exibirá uma mensagem de erro, indicando que o horário deve estar dentro do período permitido.

A segunda validação verifica se a data e hora selecionadas para a consulta são maiores que a data e hora atual, isso impede que o usuário agende consultas para o futuro, garantindo que todas as marcações sejam feitas com datas e horários atualizados. Caso a data e hora da consulta sejam maiores que a data e hora atual, o sistema exibirá uma mensagem de erro informando que isso não é permitido.

A Figura 51 representa um trecho de código referente à data da próxima consulta cadastrada no prontuário médico.

Figura 51 – Validação Data Prontuário



```
// Validar horário da próxima consulta entre 08:00 e 18:00
const proximaConsultaInput = document.querySelector('input[name="proximaConsulta"]');
const proximaConsulta = new Date(proximaConsultaInput.value);
const proximaConsultaHour = proximaConsulta.getHours();
const isProximaConsultaOutsideWorkingHours = proximaConsultaHour < 8 ||
    proximaConsultaHour >= 18;
if (isProximaConsultaOutsideWorkingHours) {
    event.preventDefault();
    isValid = false;
    errorMessage.textContent =
        'O horário da próxima consulta deve estar entre 08:00 e 18:00.';
    showMessage(errorMessage);
}

// Verificar se a data e hora da próxima consulta são maiores que a atual
if (proximaConsulta <= currentDate) {
    event.preventDefault();
    isValid = false;
    errorMessage.textContent =
        'A data e hora da próxima consulta devem ser maiores que a atual.';
    showMessage(errorMessage);
}
```

Elaborado pela autora

Neste trecho de código, estão presentes as validações para a data e horário da próxima consulta no prontuário médico. Essas validações garantem que a marcação da próxima consulta esteja dentro de um intervalo adequado e que a data e hora selecionadas sejam posteriores à data e hora atual.

A primeira validação verifica se o horário da próxima consulta está fora do horário de trabalho permitido, ou seja, se está agendado antes das 08:00 ou após as 18:00. Caso a hora da próxima consulta esteja fora desse intervalo, o sistema exibirá uma mensagem de erro indicando que o horário deve estar entre 08:00 e 18:00.

A segunda validação tem como objetivo verificar se a data e hora da próxima consulta são posteriores à data e hora atual, isso evita que o usuário agende uma próxima consulta para datas e horários já passados, garantindo que todas as marcações de consultas futuras sejam feitas corretamente. Caso a data e hora da próxima consulta sejam iguais ou anteriores à data e hora atual, o sistema exibirá uma mensagem de erro informando que a data e hora devem ser maiores que a atual.

Essas validações são importantes para garantir a confiabilidade e a precisão das marcações de consultas no prontuário médico. Ao implementar essas verificações, o sistema evita agendamentos fora do horário de trabalho e impede que consultas futuras sejam marcadas para datas e horários já passados, contribuindo para a eficiência do sistema e para a satisfação dos usuários e pacientes. Além disso, essas validações ajudam a prevenir erros e problemas futuros relacionados ao agendamento de consultas, proporcionando uma experiência mais segura e confiável para os usuários.

### 3.4.4. Campos obrigatórios

A validação na tela de agenda é feita de forma um pouco diferente das outras telas devido ao seu funcionamento específico. Quando o usuário seleciona um dia na agenda, a data de início e fim são automaticamente preenchidas com a mesma data selecionada, pois representa o agendamento para um único dia.

Em seguida, ao selecionar o horário no modal de agendamento, o horário final é definido adicionando 1 hora ao horário de início, já que as consultas têm uma duração de 1 hora por paciente. Isso é feito para garantir que o horário final seja sempre 1 hora após o horário de início, criando assim a duração de 1 hora para cada consulta agendada.

Além disso, as consultas na agenda estão disponíveis no intervalo de horários das 8h às 18h, respeitando o horário de funcionamento da clínica. Essa abordagem garante que todas as consultas sejam agendadas dentro desse período.

Essa validação específica é importante para garantir que os agendamentos sejam feitos corretamente, respeitando a duração padrão das consultas e o horário de funcionamento da clínica. Dessa forma, o sistema evita que consultas sejam marcadas com durações inconsistentes ou fora do horário de atendimento, proporcionando uma agenda organizada e adequada para o gerenciamento eficiente dos pacientes e recursos da clínica.

Além das validações mencionadas no tópico Cadastrar ou Editar um evento, também foram implementadas validações de obrigatoriedade específicas para essa tela no arquivo de *Request*, apresentadas na Figura 52.

Figura 52 – Validação Obrigatoriedade

```php
/**
 * Get the validation rules that apply to the request.
 *
 * @return array<string, mixed>
 */
public function rules()
{
    return [
        'start' => 'required',
        'end' => 'required',
        'id_paciente' => 'required',
        'id_dentista' => 'required',
        'id_clinica' => 'required',
    ];
}


public function messages()
{
    return [
        'start.required' => 'O campo horário inicial é obrigatório.',
        'end.required' => 'O campo horário final é obrigatório.',
        'id_paciente.required' => 'O campo Paciente é obrigatório.',
        'id_dentista.required' => 'O campo Dentista é obrigatório.',
        'id_clinica.required' => 'O campo Clínica é obrigatório.',
    ];
}
```

Elaborado pela autora

O trecho de código apresentado é um exemplo de como as validações de obrigatoriedade são implementadas no arquivo de *Request*. Essas validações garantem que os campos essenciais estejam preenchidos antes de prosseguir com o processamento da requisição. No método "*rules*()", são definidas as regras de validação para cada campo necessário, no caso apresentado, os campos "*start*", *"end", "id_paciente", "id_dentista"* e *"id_clinica"* são definidos como obrigatórios usando a regra "*required*".

No método "*messages*()", são definidas as mensagens de erro personalizadas que serão exibidas ao usuário caso algum dos campos obrigatórios não estejam

preenchidos. Por exemplo, se o campo start não estiver preenchido, será exibida a mensagem “O campo horário inicial é obrigatório.”.

## 3.5. Testes

Testes desempenham um papel fundamental no desenvolvimento de *software*, sendo uma prática indispensável para garantir a qualidade, confiabilidade e desempenho dos sistemas. Eles envolvem a verificação cuidadosa de cada componente e funcionalidade do software para identificar eventuais erros, bugs e falhas, assegurando que o produto final atenda aos requisitos e padrões de qualidade estabelecidos. Através de testes, é possível validar o funcionamento adequado do software, prevenir problemas futuros, facilitar a manutenção e oferecer aos usuários uma experiência satisfatória.

### 3.5.1. Abordagem utilizada

Neste contexto, a realização de testes de usabilidade é uma prática valiosa no desenvolvimento de *software*, pois permite identificar e avaliar como os usuários interagem com o sistema e como suas necessidades e expectativas são atendidas. Esses testes fornecem *insights* valiosos sobre a experiência do usuário, destacando pontos fortes e fracos da interface, bem como possíveis obstáculos ou confusões que os usuários podem enfrentar ao utilizar o *software*.

Ao conduzir testes de usabilidade, é possível coletar *feedback* direto dos usuários, permitindo que a equipe de desenvolvimento faça melhorias e ajustes no design e na interface do sistema para torná-lo mais intuitivo, eficiente e amigável. Isso resulta em uma experiência mais agradável para os usuários, o que, por sua vez, contribui para a satisfação do cliente, fidelização e até mesmo para a aquisição de novos usuários através de recomendações positivas.

Os testes de usabilidade podem ser realizados através de diferentes abordagens, como testes de protótipos interativos, observação de usuários em situações reais de uso, questionários de satisfação, entre outros métodos. Essas avaliações fornecem uma visão holística do sistema e da interação do usuário com

ele, permitindo que a equipe de desenvolvimento tome decisões informadas para aprimorar o produto final.

Utilizamos uma abordagem de teste que consistiu em simular cenários diários de uma clínica, permitindo-nos experimentar como o sistema seria utilizado na prática. Essa metodologia nos proporcionou uma visão realista de como o sistema se comporta sob condições reais de uso, possibilitando avaliar sua eficiência e usabilidade em situações cotidianas. Ao passar por esses cenários diários simulados, pudemos explorar diversas funcionalidades do sistema, desde o agendamento de consultas até o registro de prontuários, analisando como as ações e processos fluíam de maneira intuitiva e eficiente.

### 3.5.2. Cenários simulados

Neste contexto, apresentaremos os principais cenários simulados e discutiremos como esses testes contribuíram para aprimorar a qualidade e a usabilidade do sistema, garantindo uma experiência positiva tanto para os profissionais da área odontológica quanto para os pacientes:

- Agendamento de Consulta: Simular o agendamento de uma consulta para um paciente específico, escolhendo um dentista com horários disponíveis e uma clínica. Verificar se a data e hora selecionadas estão dentro dos intervalos válidos e se a consulta é corretamente registrada no sistema.
- Cadastro de Paciente: Simular o cadastro de um novo paciente na clínica, preenchendo todas as informações obrigatórias, como nome, data de nascimento, *CPF* e endereço. Verificar se todas as validações de campos são aplicadas corretamente.
- Registro de Prontuário: Simular o registro de prontuários para pacientes após as consultas. Inserir informações relevantes, como diagnósticos, procedimentos realizados e prescrições médicas. Verificar se todas as informações são salvas corretamente no sistema.
- Cancelamento de Consulta: Simular o cancelamento de uma consulta agendada, verificando se o sistema atualiza corretamente a agenda do dentista e da clínica, liberando o horário para novos agendamentos.

- Verificação de Conflitos: Criar cenários em que dois ou mais pacientes tentam agendar uma consulta no mesmo horário com o mesmo dentista ou na mesma clínica. Verificar se o sistema consegue detectar e evitar conflitos de agendamento.
- Atualização de Cadastros: Testar a capacidade do sistema de atualizar informações de pacientes, dentistas e clínicas, garantindo que as alterações sejam refletidas corretamente em todo o sistema.
- Gerenciamento de Usuários: Simular o processo de adicionar, remover e modificar os usuários que têm acesso ao sistema, garantindo que apenas usuários autorizados tenham permissão para acessar determinadas funcionalidades.

## 3.6. Resolução de problemas e ajustes

Os testes realizados desempenharam um papel crucial na identificação de pontos de melhoria no sistema, resultando em uma experiência mais satisfatória para os usuários. Antes da implementação das validações para os horários dos dentistas, o agendamento de consultas era feito manualmente, permitindo sobreposição de horários e agendamentos simultâneos para o mesmo profissional.

Contudo, graças aos testes realizados, foi possível reconhecer essa questão e implementar validações que garantem a seleção de horários padronizados a partir de um seletor pré-definido. Com essa atualização, o sistema evita agendamentos conflitantes, impedindo que mais de um paciente seja marcado para o mesmo horário do dentista. Essa medida aprimora a organização dos atendimentos e aumenta a eficiência na marcação de consultas, resultando em uma experiência mais fluida e confiável para todos os envolvidos na clínica odontológica.

Além disso, foi identificado um problema no horário de fechamento da clínica, que antes era permitido agendar até as 19h, comprometendo a organização dos atendimentos e a disponibilidade do profissional. A partir dos testes, foi possível reconhecer esse cenário e realizar a modificação necessária, estabelecendo o último horário de início de consultas às 17h e a conclusão desses agendamentos às 18h, assegurando que todos os pacientes sejam atendidos dentro do horário de funcionamento da clínica. Essas melhorias não apenas garantiram a integridade do

sistema, mas também contribuíram para a otimização dos processos de agendamento e para a satisfação tanto dos profissionais da área odontológica quanto dos pacientes.

Os testes foram igualmente essenciais para identificar aprimoramentos necessários nos campos de data do prontuário. Anteriormente, o sistema permitia a seleção de horários fora do horário de funcionamento da clínica, o que poderia causar cadastros incorretos. Com base nos resultados dos testes, foram implementadas modificações e validações para garantir que não houvesse datas de próximas consultas menores que a atual, bem como datas de consulta maiores que a data atual. Além disso, foram aplicadas restrições para que os horários selecionados estejam dentro do período de funcionamento da clínica, garantindo assim que todas as consultas estejam agendadas dentro dos horários disponíveis para atendimento.

Com o sistema devidamente validado e adaptado às necessidades reais, a clínica odontológica agora desfruta de um ambiente mais confiável, eficiente e alinhado com suas demandas cotidianas, proporcionando uma experiência mais satisfatória e fluida para todos os envolvidos.

# 4. Análise dos Resultados e Avaliação do Sistema

## 4.1. Telas da aplicação

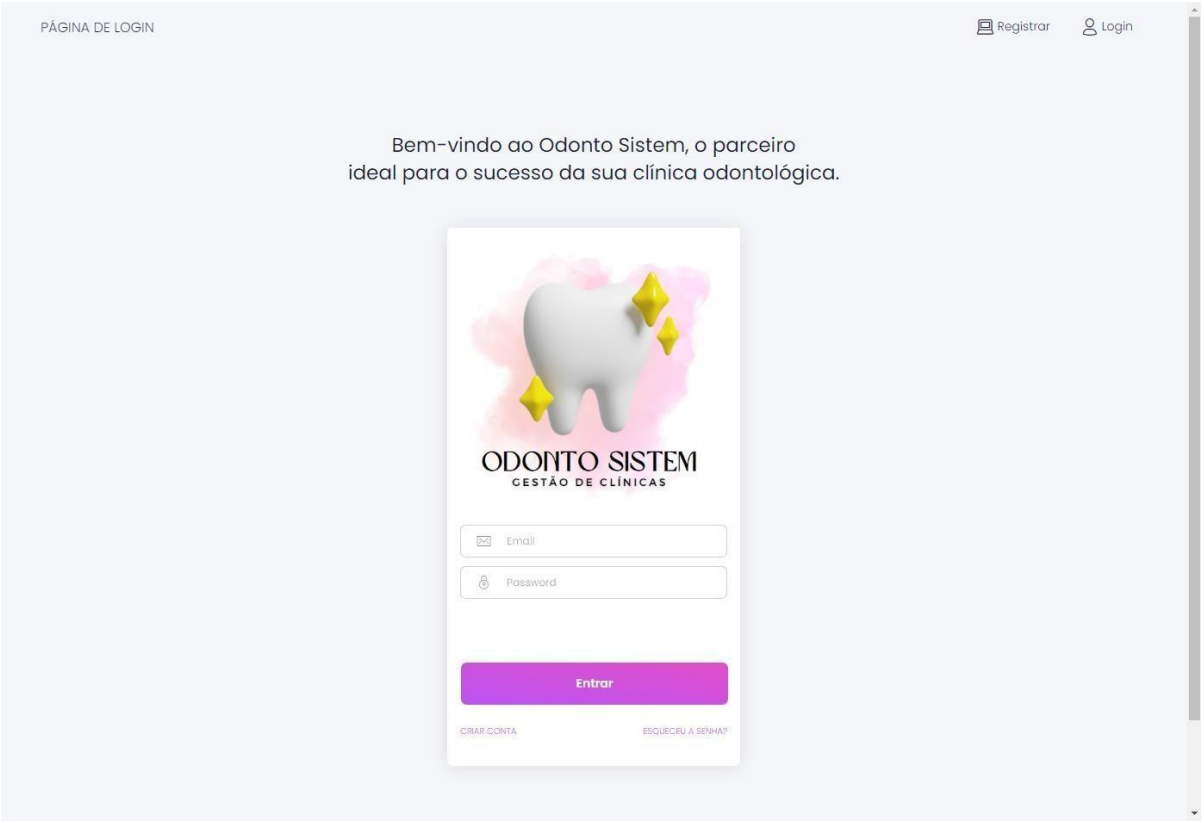
A seguir, apresentarei uma breve análise dos resultados obtidos após o desenvolvimento do sistema, destacando o desempenho e as funcionalidades de cada tela. Cada tela passou por um processo de testes, simulações e ajustes, buscando aprimorar a experiência do usuário e garantir a eficiência do sistema como um todo. Serão descritos os principais resultados de cada tela, ressaltando os avanços em termos de usabilidade, layout intuitivo e funcionalidades implementadas.

### 4.1.1. Tela de Login/Registrar

#### 4.1.1.1. Login

A Figura 53 representa a tela de *login* do sistema, apresentando uma interface intuitiva para os usuários. No canto superior direito, há botões de fácil acesso para o registro de novos usuários, tornando o processo de criação de contas simples e rápido. No centro da tela, encontra-se o formulário de *login*, onde os usuários podem inserir suas credenciais, como o e-mail e a senha, após preencher os campos corretamente, o usuário pode utilizar o botão “Entrar” para acessar o sistema. Além disso, são oferecidas opções abaixo do botão de *login* para criar uma nova conta ou redefinir a senha, proporcionando aos usuários uma experiência mais completa e prática. Essa tela é projetada para facilitar o acesso ao sistema e fornecer as opções necessárias para os usuários gerenciarem suas contas de forma eficiente.

Figura 53 – Tela de login

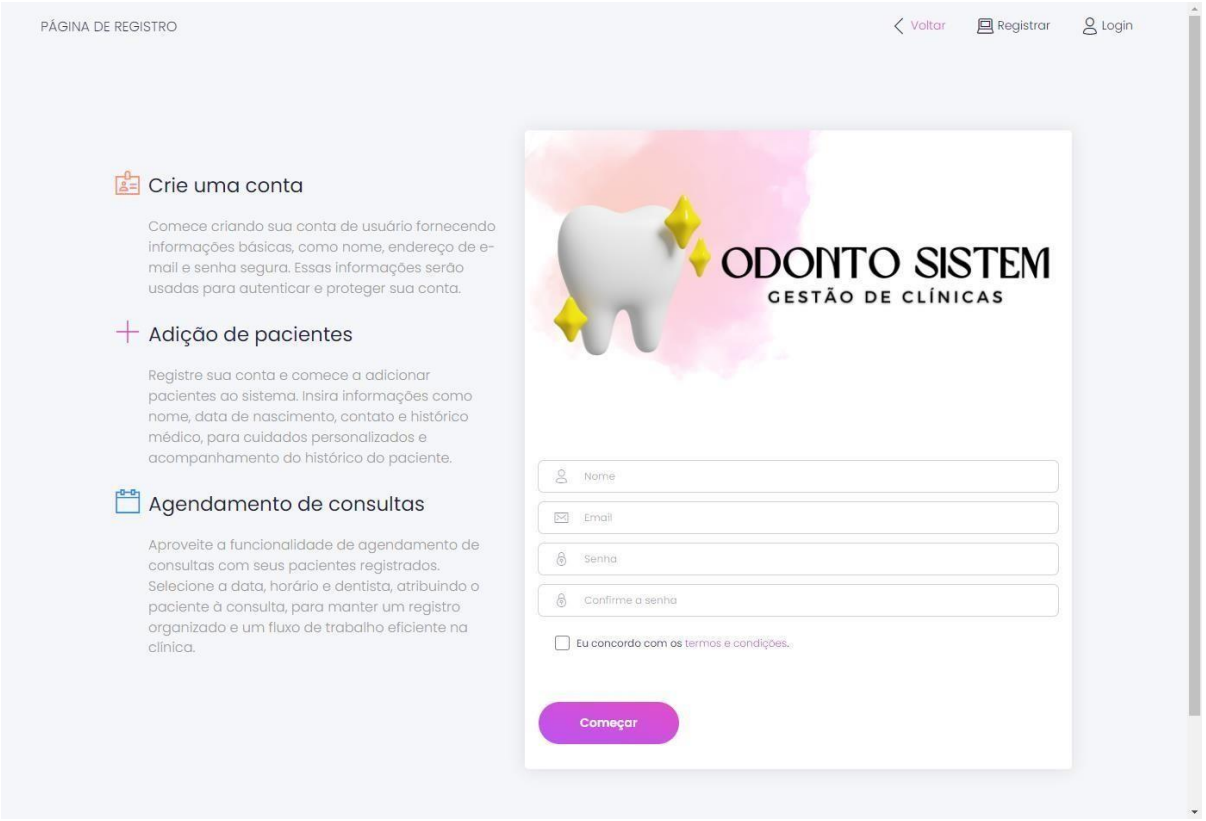



Elaborado pela autora

#### 4.1.1.2. Registrar

A Figura 54 representa a tela de registro de usuário do sistema, oferecendo uma experiência amigável e acessível. No canto superior direito, encontram-se botões de fácil acesso para realizar o *login* ou voltar à tela anterior. Ao centro, há informações sobre o sistema, proporcionando uma breve apresentação aos usuários. Ao lado dessas informações, está o formulário de registro, onde os usuários podem preencher seus dados, incluindo nome, e-mail, senha e confirmação de senha. Abaixo do formulário, há um *checkbox* para aceitar os termos e condições do sistema, garantindo o consentimento do usuário, após preencher corretamente os campos e aceitar os termos, o usuário pode utilizar o botão “Começar” para criar sua conta no sistema.

Figura 54 – Tela de registro



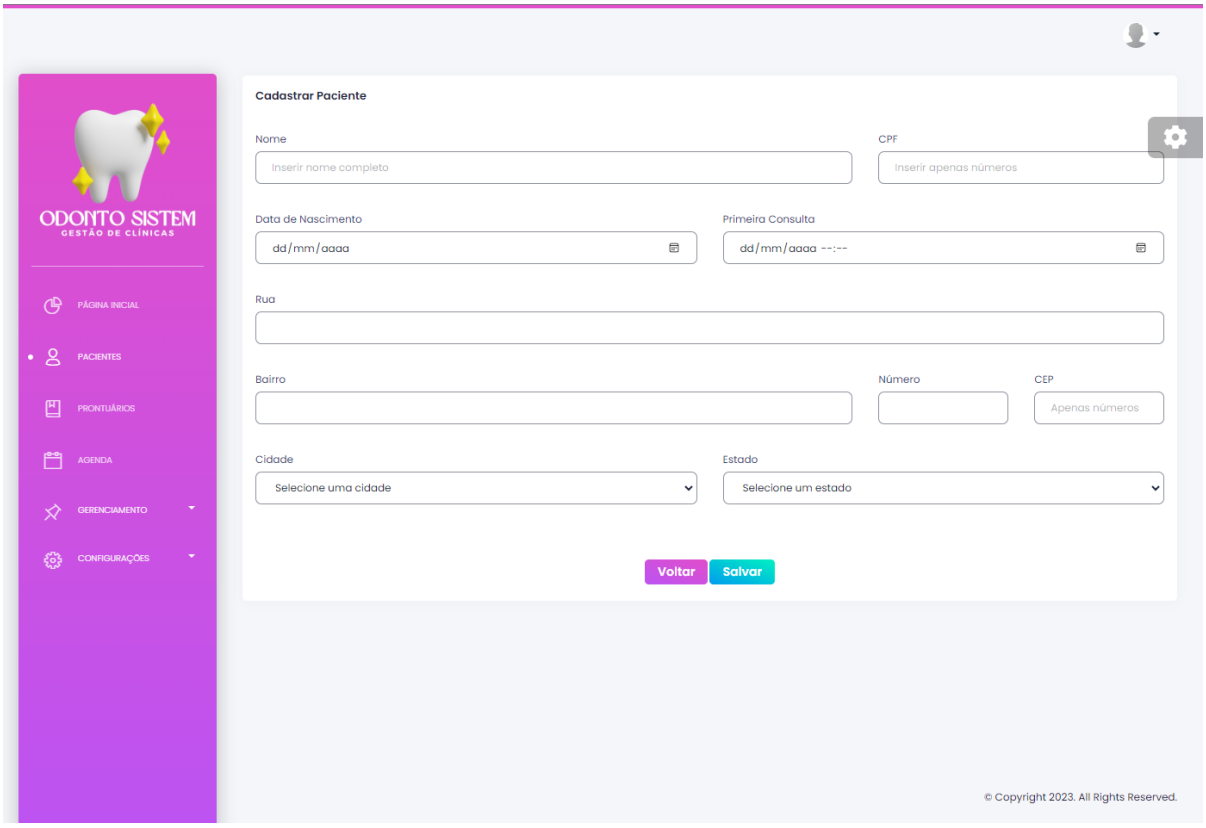

Elaborado pela autora

### 4.1.2. Tela de Pacientes

#### 4.1.2.1. Cadastro

A Figura 55 representa a tela de cadastro de paciente, onde são inseridos dados como nome, *CPF*, data de nascimento, primeira consulta e informações de endereço. A visualização padronizada e a disposição lógica dos campos facilitam o preenchimento correto e completo das informações do paciente, tornando o processo de cadastro mais ágil e eficiente para as equipes da clínica. Os botões “Voltar” e “Salvar” permitem retornar à tela anterior ou confirmar e salvar os dados inseridos, respectivamente.

Figura 55 – Tela cadastro pacientes

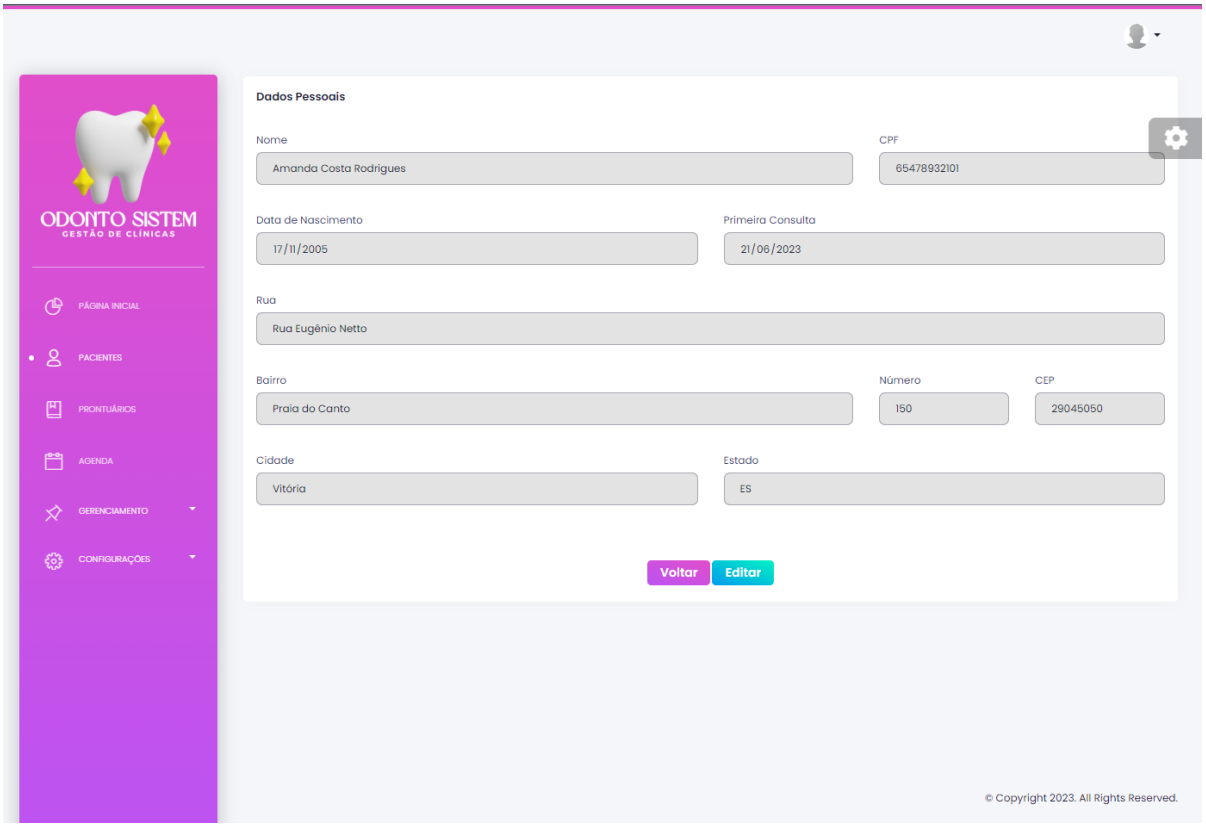



Elaborado pela autora

### 4.1.2.2. Edição

A Figura 56 retrata a tela de edição de paciente, com um *layout* idêntico ao das telas de cadastro e visualização, nessa tela, todos os campos estão bloqueados para evitar alterações indesejadas. Esse padrão é seguido em todas as telas de cadastro, edição e visualização para manter a consistência e facilitar a usabilidade do sistema. A diferença entre essas telas reside nos botões localizados na parte inferior, na tela de edição, são exibidos os botões “Voltar”, “Salvar” e “Deletar”. O botão “Voltar” permite retornar à tela anterior, “Salvar” possibilita confirmar as alterações feitas nos dados do paciente, e “Deletar” oferece a opção de remover permanentemente o registro, tornando o gerenciamento dos pacientes mais prático e intuitivo.

Figura 56 – Tela edição pacientes

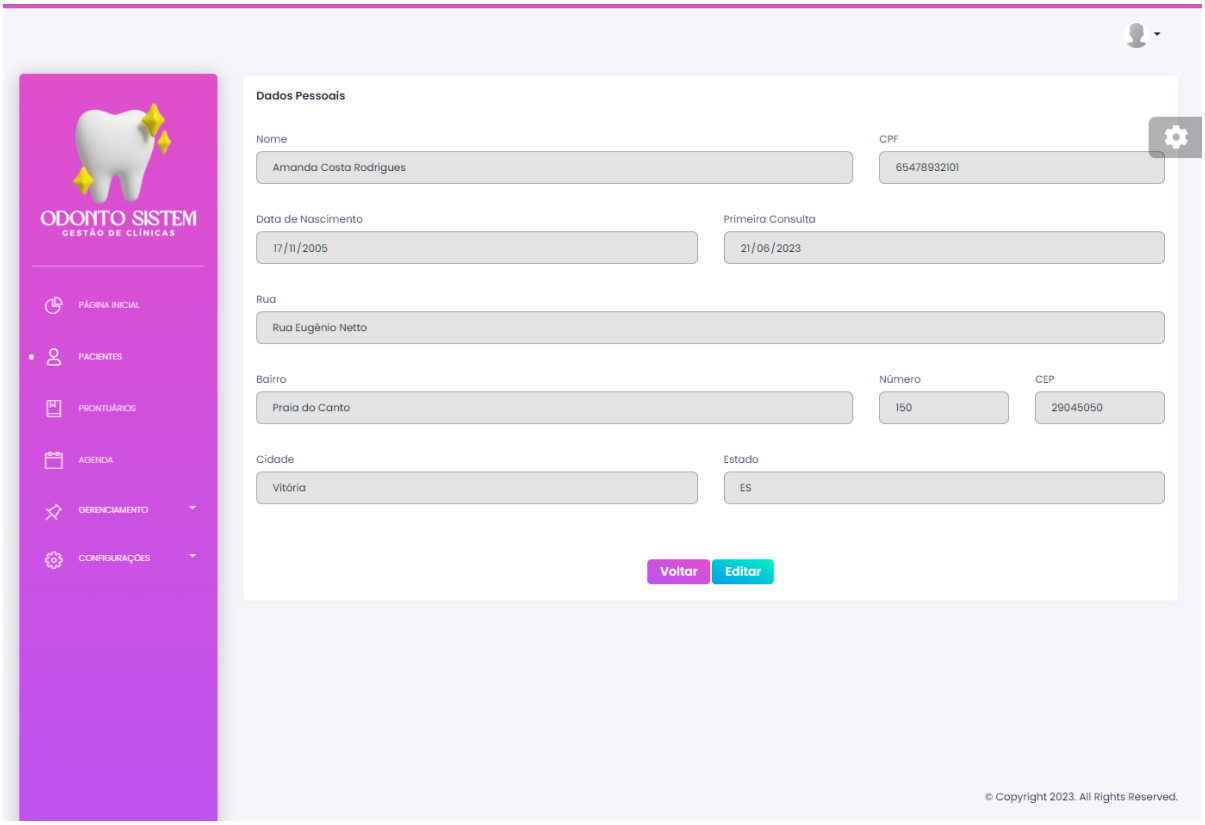



Elaborado pela autora

#### 4.1.2.3. Visualização

A Figura 57 representa a tela de visualização de paciente. Essa tela possui o mesmo *layout* das telas de cadastro e edição, porém todos os campos estão bloqueados, impedindo a alteração dos dados. Essa abordagem garante que as informações do paciente sejam visualizadas de forma precisa, sem riscos de alterações acidentais. A diferença entre essas telas está nos botões localizados ao final da página. Na tela de visualização, encontramos os botões “Voltar” e “Editar”. O botão “Voltar” permite retornar à tela anterior, enquanto o botão “Editar” desbloqueia os campos, possibilitando a edição dos dados, proporcionando uma experiência intuitiva e facilitando o gerenciamento dos pacientes no sistema.

Figura 57 – Tela visualização de pacientes

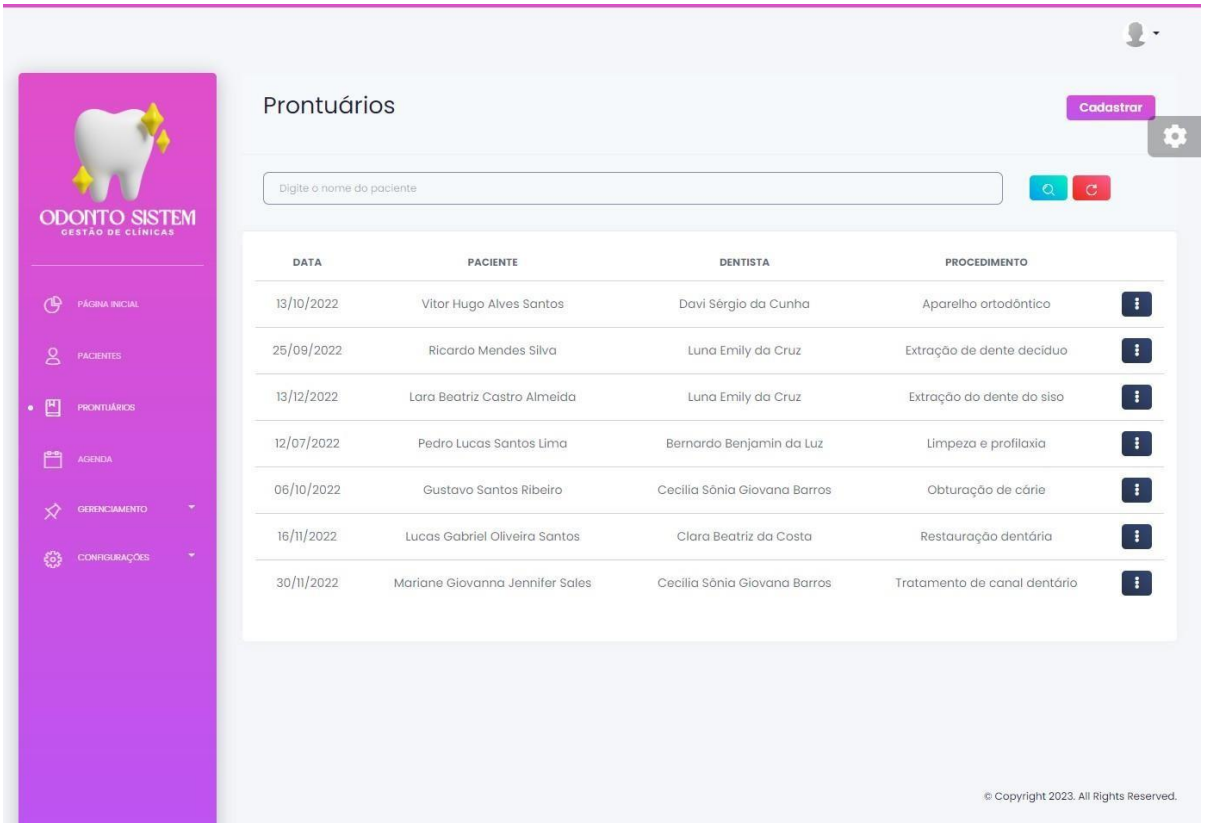



Elaborado pela autora

### 4.1.3. Tela de Prontuários

A Figura 58 exibe a tela de prontuários, que é composta por um botão de cadastro na parte superior, seguido por uma barra de busca que ocupa toda a largura da página, permitindo ao usuário pesquisar prontuários pelo nome do paciente. Logo abaixo, é apresentada uma tabela contendo informações sobre os prontuários, incluindo a data, o nome do paciente, o dentista responsável e a data do procedimento. Ao lado de cada registro na tabela, há um botão “Visualizar”, que permite ao usuário acessar detalhes específicos do prontuário. Dentro da visualização, o usuário tem a opção de editar ou deletar o registro, proporcionando maior controle sobre os dados dos prontuários de forma intuitiva e eficiente.

Figura 58 – Tela de Prontuários



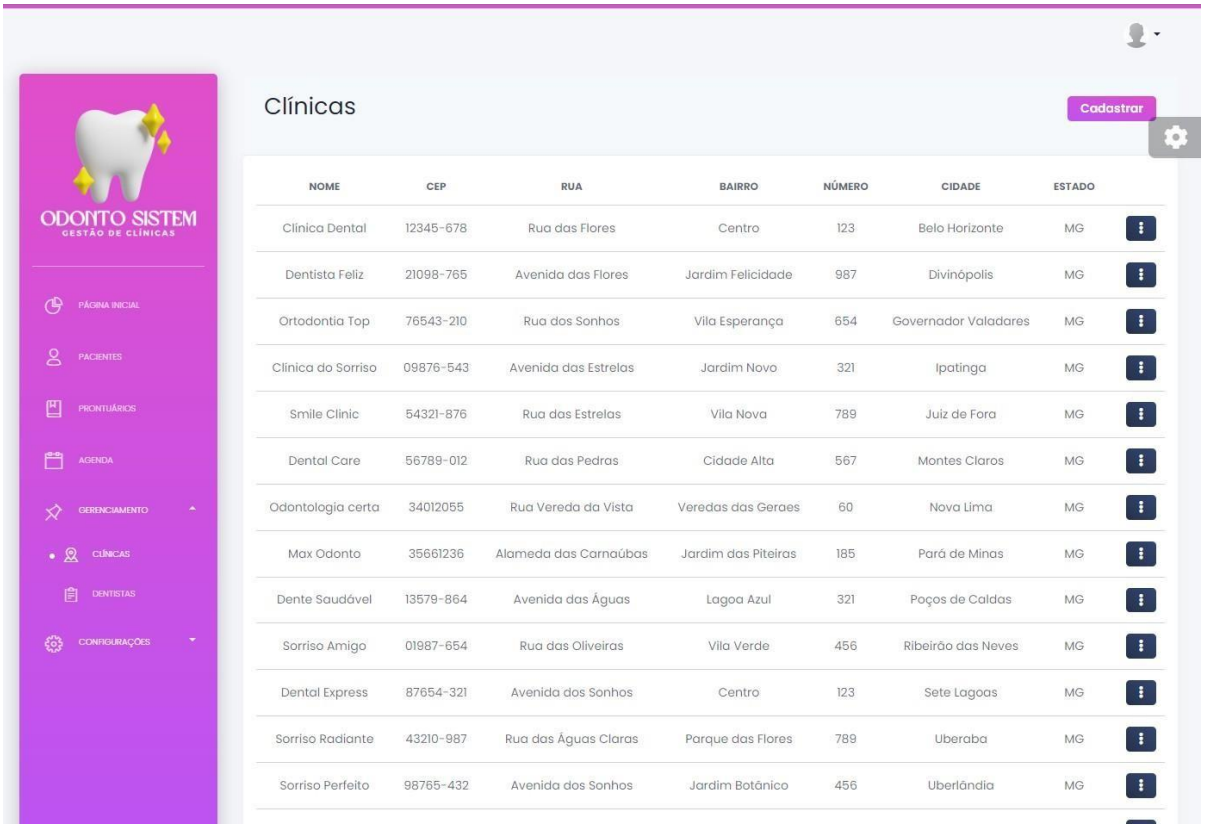

Elaborado pela autora

### 4.1.4. Telas de Gerenciamento

#### 4.1.4.1. Clínicas

A Figura 59 ilustra a tela de clínicas, que possui um botão de cadastro no canto superior direito. Abaixo desse botão, há uma tabela com todas as clínicas cadastradas no sistema, apresentando informações como nome, *CEP*, rua, bairro, número, cidade e estado. Cada registro da tabela possui um botão de edição, ao clicar no botão de edição, é possível acessar a tela de edição da clínica, onde o usuário pode fazer alterações nos dados e também tem a opção de deletar a clínica.

Figura 59 – Tela de Clínicas

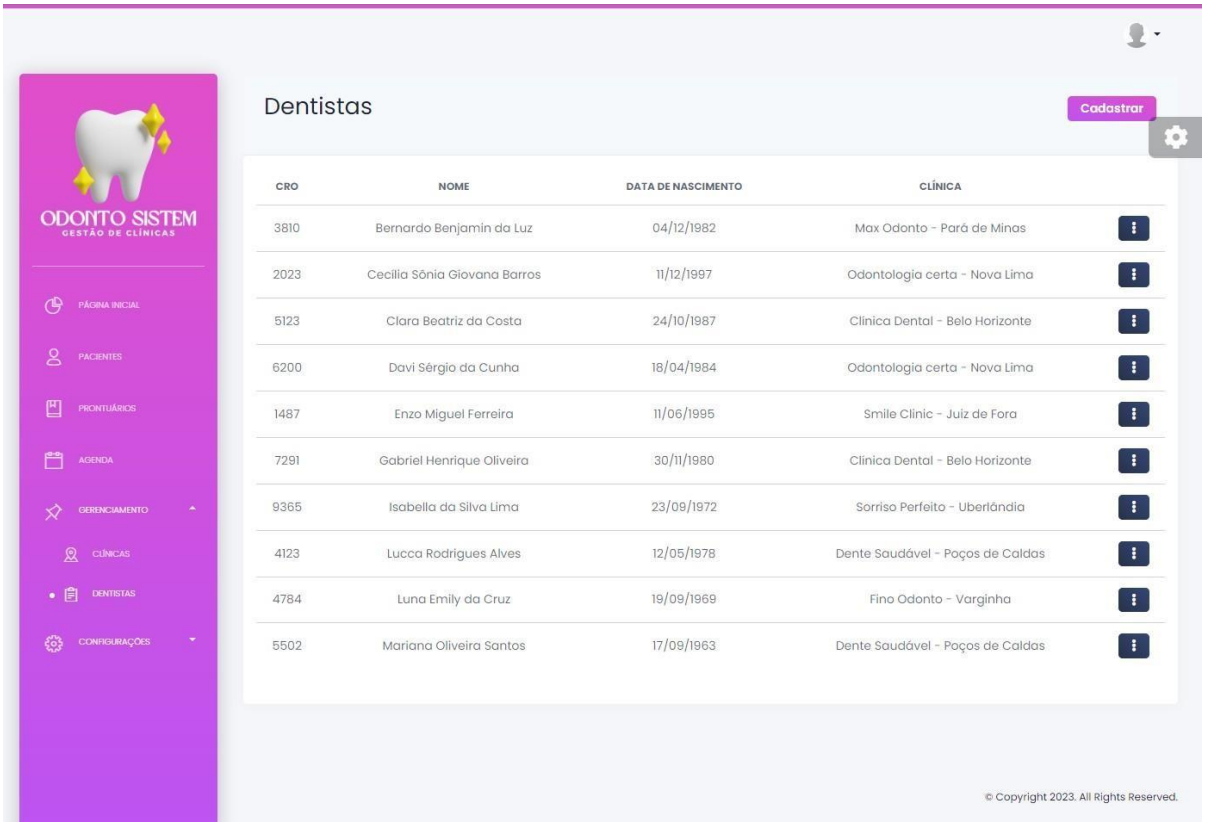



Elaborado pela autora

#### 4.1.4.2. Dentistas

A Figura 60 representa a tela de dentistas, que conta com um botão de cadastro no canto superior direito. Abaixo desse botão, há uma tabela com todos os dentistas cadastrados no sistema, exibindo informações como *CRO* (Cadastro Regional de Odontologia), nome, data de nascimento e clínica associada. Cada registro na tabela inclui um botão para visualização, e ao clicar nesse botão, é possível acessar a tela de visualização do dentista, na tela de visualização, o usuário tem a opção de editar os dados do dentista ou deletá-lo do sistema.

Figura 60 – Tela de Dentistas



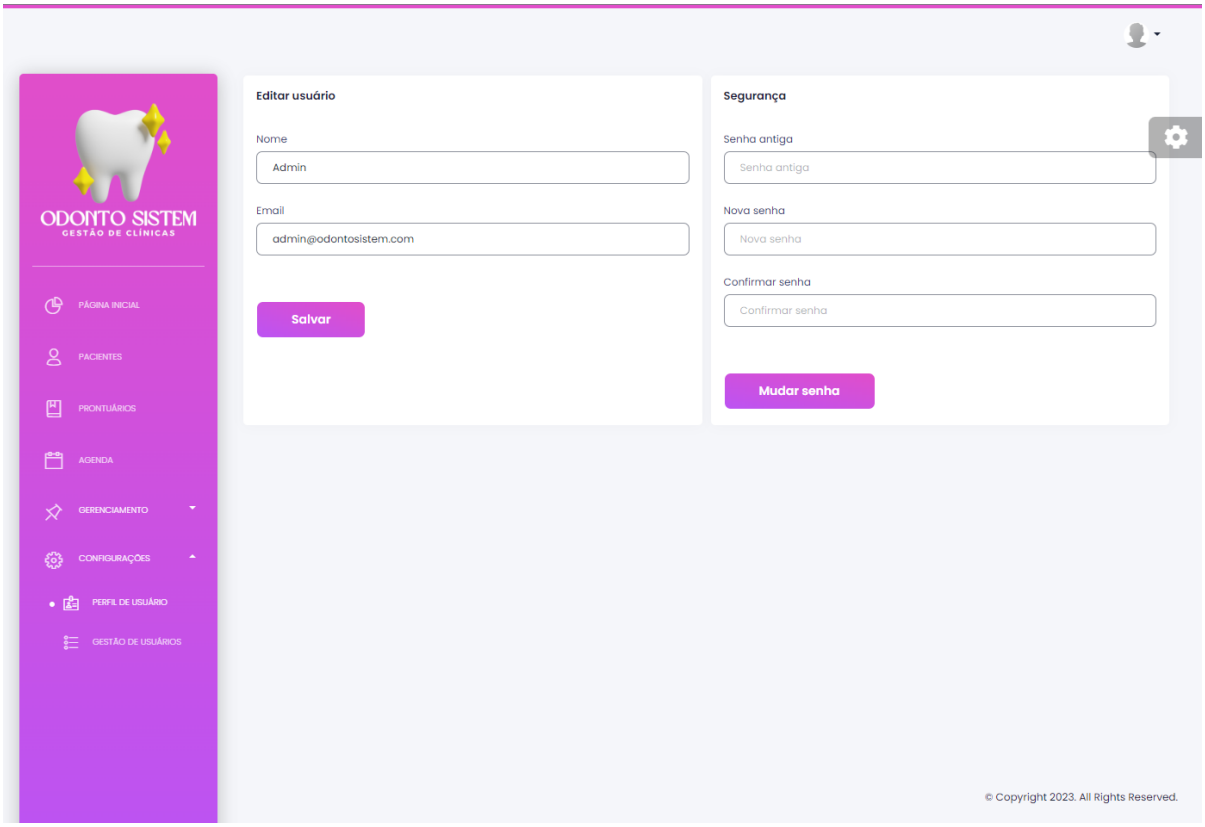

Elaborado pela autora

### 4.1.5. Telas de Configurações

#### 4.1.5.1. Perfil de usuário

A Figura 61 apresenta a tela de perfil de usuário, que oferece duas seções importantes para edição e segurança. Na primeira seção, o usuário pode editar seu nome e e-mail, permitindo manter suas informações atualizadas, já na segunda seção, é possível modificar a senha, garantindo a segurança da conta. Para alterar a senha, o usuário deve fornecer a senha antiga como confirmação, seguida do novo conjunto de senha e sua confirmação. Por fim, são disponibilizados os botões “Salvar” e “Mudar Senha”, permitindo que o usuário aplique as alterações em seu perfil ou em sua senha, respectivamente.

Figura 61 – Tela Perfil de Usuário

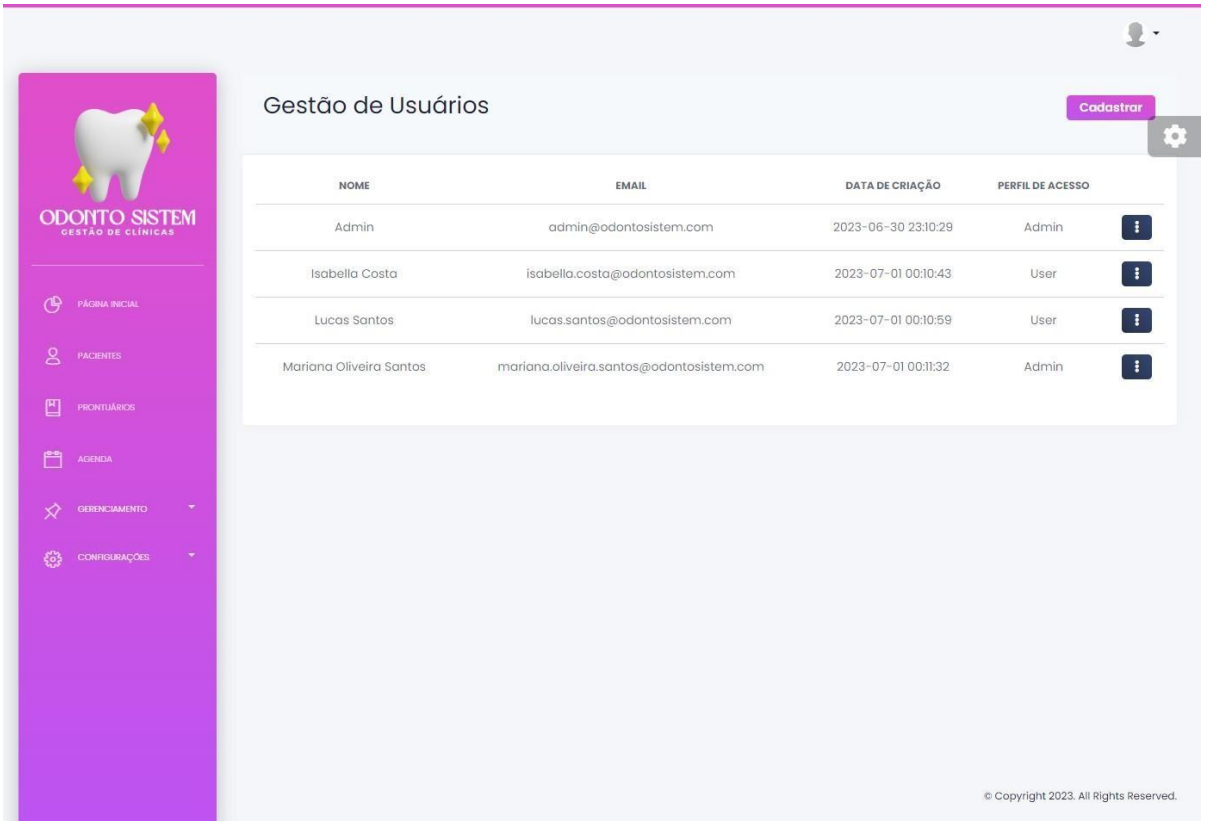



Elaborado pela autora

#### 4.1.5.2. Gestão de usuários

A Figura 62 representa a tela de gestão de usuários, onde os administradores do sistema podem gerenciar os usuários cadastrados. No canto superior direito, há um botão para cadastrar novos usuários, facilitando a inclusão de novos membros, logo abaixo, encontra-se uma tabela contendo todos os usuários cadastrados, exibindo informações importantes como nome, e-mail, data de criação e perfil de acesso, que pode ser definido como “*user*” ou “*admin*”. Cada registro na tabela possui um botão “Editar”, permitindo aos administradores realizarem alterações nas informações dos usuários. Ao selecionar a opção “Editar”, a tela de edição é exibida, oferecendo a possibilidade de editar os dados do usuário cadastrado, bem como a opção de “Deletar”, para remover o usuário, se necessário.

Figura 62 – Tela Gestão de Usuários




Elaborado pela autora

## 4.2. Considerações finais

No Capítulo 4, foram apresentados os resultados do estudo e desenvolvimento do projeto, abrangendo a descrição das telas e funcionalidades da aplicação desenvolvida. Ao longo desse capítulo, foi possível demonstrar como a plataforma atende aos objetivos propostos, destacando sua eficiência e funcionalidade na gestão de clínicas odontológicas. No Capítulo 5, traremos a conclusão geral deste trabalho, abrangendo também novas possibilidades para o aprimoramento contínuo da plataforma.

# 5. Conclusões

O presente trabalho teve como objetivo o desenvolvimento de uma plataforma de gerenciamento para clínicas odontológicas, utilizando o *framework Laravel*, seguindo o padrão de projeto *MVC* e interagindo com o banco de dados *MySQL*. O objetivo geral foi alcançado com sucesso, e os objetivos específicos foram cumpridos, proporcionando uma solução completa e eficiente para a gestão das clínicas, pacientes, dentistas e prontuários.

Uma das principais contribuições alcançadas foi a criação de uma plataforma robusta e segura, que atende às necessidades específicas do setor odontológico. A utilização do *framework Laravel* permitiu o desenvolvimento ágil e elegante da aplicação, enquanto o padrão de projeto *MVC* possibilitou uma organização clara e uma manutenção mais fácil do código. A representação visual dos dados através de gráficos interativos foi outro ponto relevante, proporcionando uma compreensão rápida e intuitiva das informações de desempenho de atendimentos ao longo do tempo e da distribuição de pacientes por estado. Isso contribuiu para uma análise mais eficiente das informações e auxiliou na tomada de decisões estratégicas.

A interface de usuário intuitiva e amigável foi um dos destaques do trabalho, oferecendo uma experiência agradável aos usuários e possibilitando uma interação fluida com as funcionalidades do sistema, além disso, a plataforma apresenta uma estrutura completa de *CRUD* para os cadastros, proporcionando uma gestão completa dos dados, desde a criação até a edição e exclusão de registros. Contudo, é importante destacar algumas limitações do estudo, a plataforma desenvolvida foi testada em um ambiente controlado, e embora tenha sido projetada com foco na segurança, é essencial realizar uma análise mais aprofundada de vulnerabilidades e realizar testes de segurança para garantir a proteção dos dados.

Em conclusão, o trabalho alcançou os objetivos propostos ao desenvolver uma plataforma de gerenciamento de clínicas odontológicas completa, segura e intuitiva, utilizando tecnologias modernas e melhores práticas de desenvolvimento. A aplicação resultante proporciona aos profissionais da área odontológica uma ferramenta eficiente e confiável para a gestão de suas atividades, melhorando a qualidade do atendimento e contribuindo para o sucesso de suas clínicas.

## 5.1. Trabalhos futuros

As limitações do escopo do projeto revelaram importantes oportunidades de trabalhos futuros, que podem aprimorar ainda mais a plataforma desenvolvida. Uma das possibilidades é a integração da plataforma com outros sistemas ou serviços, como sistemas de agendamento online, gestão financeira ou comunicação com pacientes, essas integrações permitiriam uma experiência mais completa e conveniente para os usuários, possibilitando uma gestão mais abrangente e eficiente das clínicas odontológicas.

Outra melhoria significativa seria a inclusão de recursos adicionais na plataforma, como por exemplo, a possibilidade de enviar notificações aos pacientes sobre consultas agendadas pode contribuir para uma maior eficiência e redução de faltas, garantindo um fluxo mais organizado de atendimentos. O desenvolvimento de aplicativos móveis também seria uma adição valiosa, facilitando o acesso à plataforma por meio de dispositivos móveis, o que é especialmente relevante em um contexto cada vez mais *mobile*. Ademais, a incorporação de recursos de gestão financeira pode oferecer aos profissionais de odontologia uma visão mais abrangente e detalhada das finanças das clínicas, permitindo um melhor controle de receitas, despesas e fluxo de caixa.

Essas melhorias propostas possibilitariam a criação de um sistema mais automatizado e inteligente, capaz de se adaptar às necessidades e demandas em constante evolução do setor odontológico. É importante ressaltar que, ao implementar essas melhorias e trabalhos futuros, é fundamental manter a preocupação contínua com a segurança dos dados, medidas adicionais devem ser implementadas para garantir a privacidade e a proteção dos dados dos pacientes, assegurando a confiança dos usuários na plataforma.

Em suma, as oportunidades de trabalhos futuros identificadas revelam o potencial de evolução e expansão da plataforma, tornando-a uma ferramenta ainda mais relevante e valiosa para o setor odontológico. Ao considerar essas possibilidades, o trabalho apresentado aqui pode servir como uma base sólida para o avanço tecnológico no campo da odontologia, contribuindo para o aprimoramento do atendimento e da gestão das clínicas odontológicas no futuro.

## Referências

BENTO, Evaldo Junior. Desenvolvimento web com PHP e MySQL. Editora Casa do Código, 2021.

HARTMANN, Frederico G.; CANTARELLI, Gustavo Stangherlin. Sistema de Gestão para Clínicas Odontológicas.

INAN, Dedi Iskandar; JUITA, Ratna. Analysis and design complex and large data base using MySQL workbench. International Journal of Computer Science & Information Technology, v. 3, n. 5, p. 173, 2011.

LAAZIRI, Majida et al. A comparative study of laravel and symfony PHP frameworks. International Journal of Electrical and Computer Engineering, v. 9, n. 1, p. 704-712, 2019.

LAAZIRI, M., El Bouhdidi, J., & El Mohajir, B. (2019). A Comparative Study of PHP Frameworks for Web Application Development. Procedia Manufacturing, 32, 864-871. https://doi.org/10.1016/j.promfg.2019.02.119

LARAVEL. Laravel Documentation. Disponível em: https://laravel.com/docs/9.x/readme . Acesso em: 02 julho 2023.

LUCIANO, Josué; ALVES, Wallison Joel Barberá. Padrão de arquitetura MVC: Model-view-controller. EPeQ Fafibe, v. 1, n. 3a, p. 102-107, 2017.

MYSQL. MySQL Workbench Documentation. Disponível em: https://dev.mysql.com/doc/refman/8.0/en/ . Acesso em: 02 julho 2023.

NOGUEIRA, João Danilo. Linguagem de Programação para Web - PHP. 1ª ed. São Paulo: Editora XPTO, 2022. Disponível em: http://18.222.122.60/lpw/fael/4/un4/ebook.pdf . Acesso em: 30 de junho de 2023.

RICHARD, Guy. A Framework Comparison:. NET and Laravel. 2022.

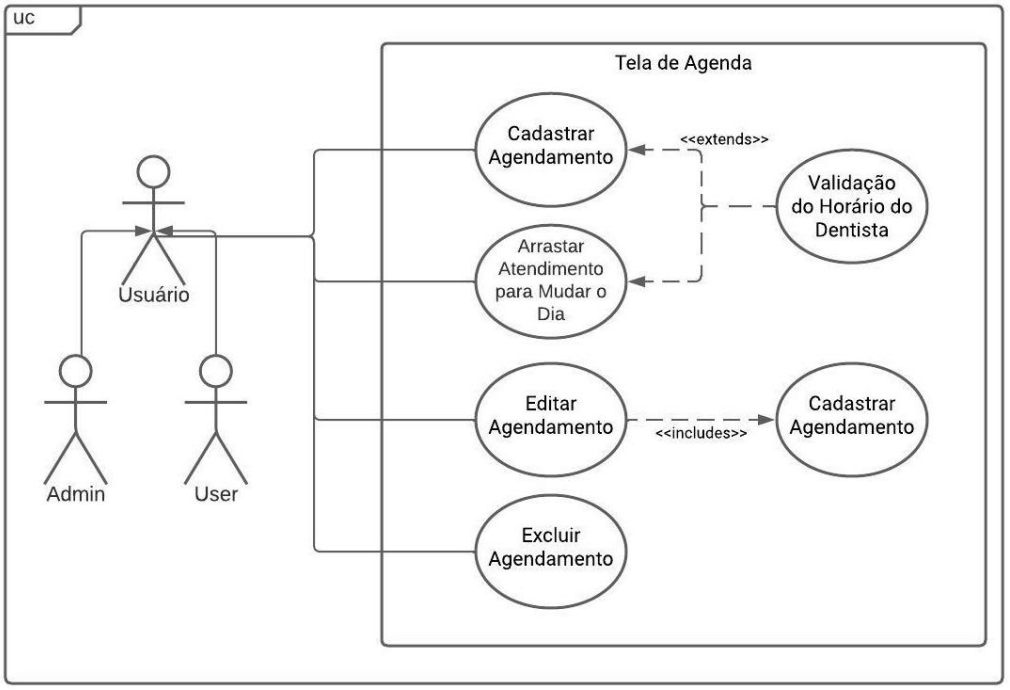
SIMDOCTOR. Vantagens do software odontológico para o dia a dia do dentista. 31 de julho de 2018. Disponível em: https://www.simdoctor.com.br/blog/vantagens-do-software-odontologico/ . Acesso em: 02 julho 2023.

# APÊNDICES

## APÊNDICE A – Diagramas de casos de uso

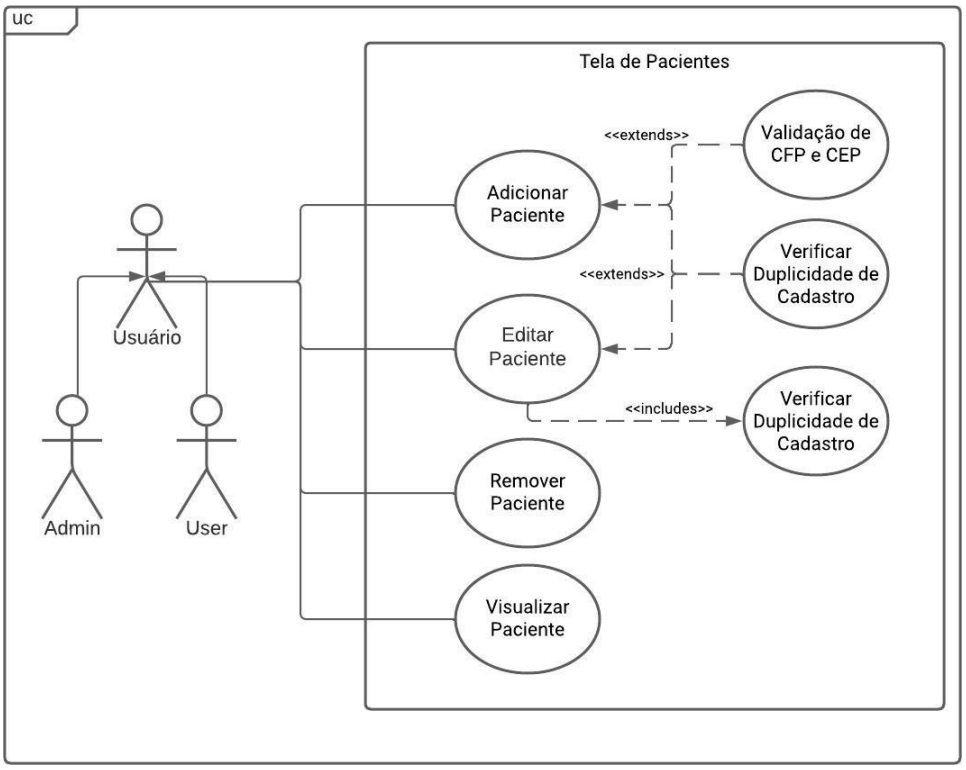
A Figura 63 ilustra o caso de uso do módulo de agenda. A Figura 64 ilustra o caso de uso do módulo de pacientes. A Figura 65 ilustra o caso de uso do módulo de perfil de usuário. A Figura 66 ilustra o caso de uso do módulo de dentista. A Figura 67 ilustra o caso de uso do módulo de clínica. A Figura 68 ilustra o caso de uso do módulo de prontuário. A Figura 69 ilustra o caso de uso do módulo de gestão de usuários. A Figura 70 ilustra o caso de uso do módulo de painel inicial.

Figura 63 - Diagramas de Casos de Uso – Agenda



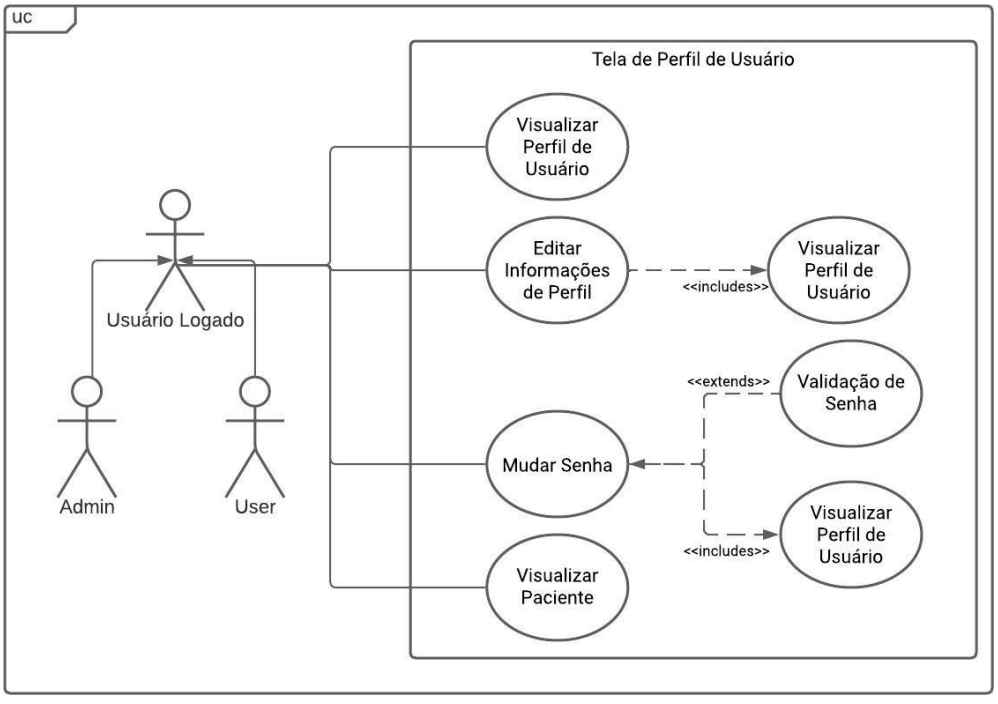

Elaborado pela autora

Figura 64 - Diagramas de Casos de Uso - Pacientes



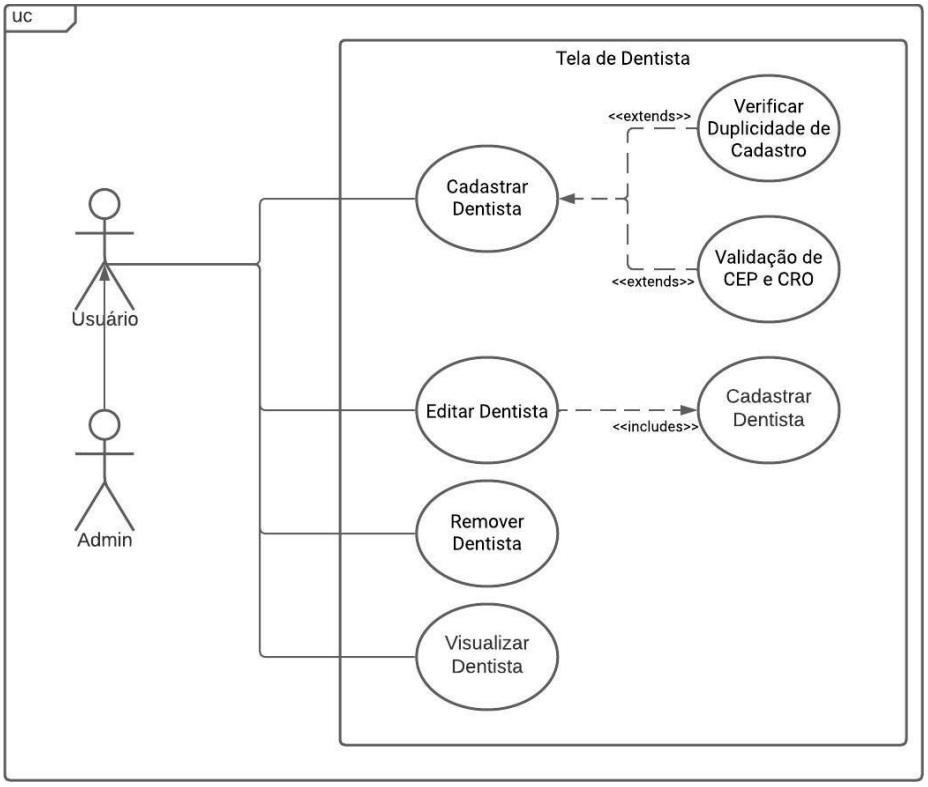

Elaborado pela autora

Figura 65 - Diagramas de Casos de Uso - Perfil de Usuário



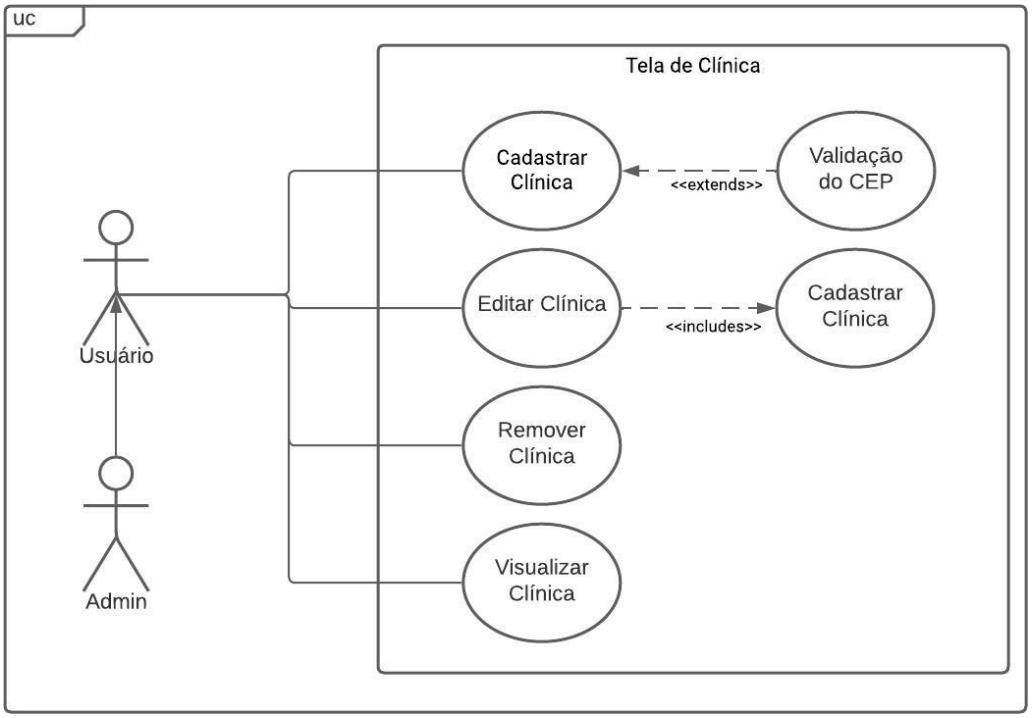

Elaborado pela autora

Figura 66 - Diagramas de Casos de Uso – Dentista



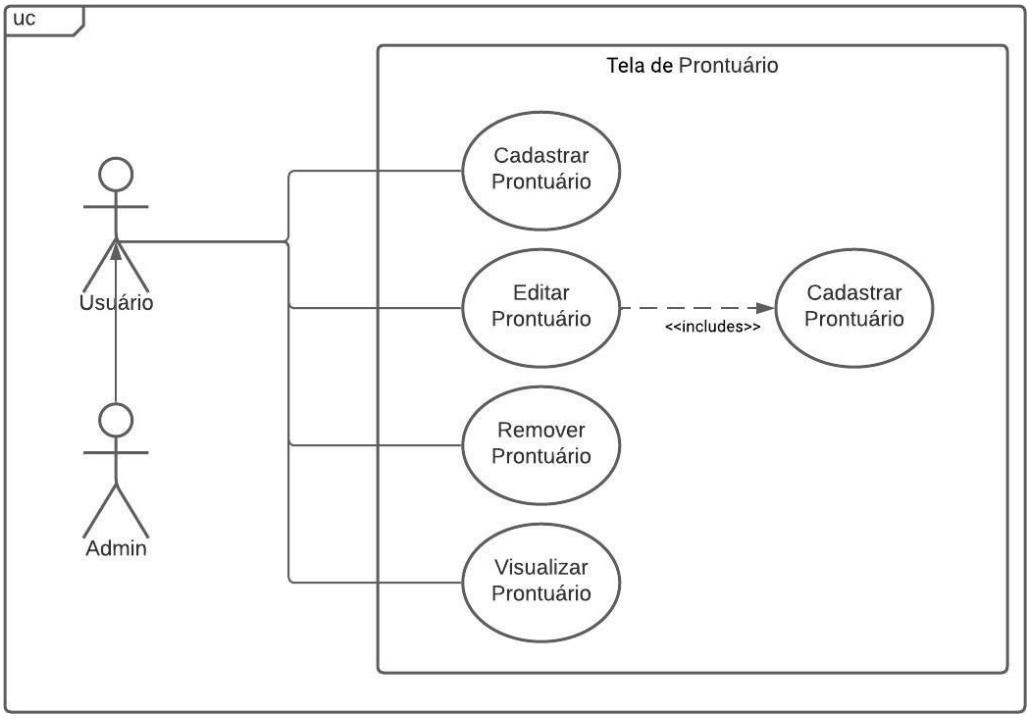

Elaborado pela autora

Figura 67 - Diagramas de Casos de Uso - Clínica



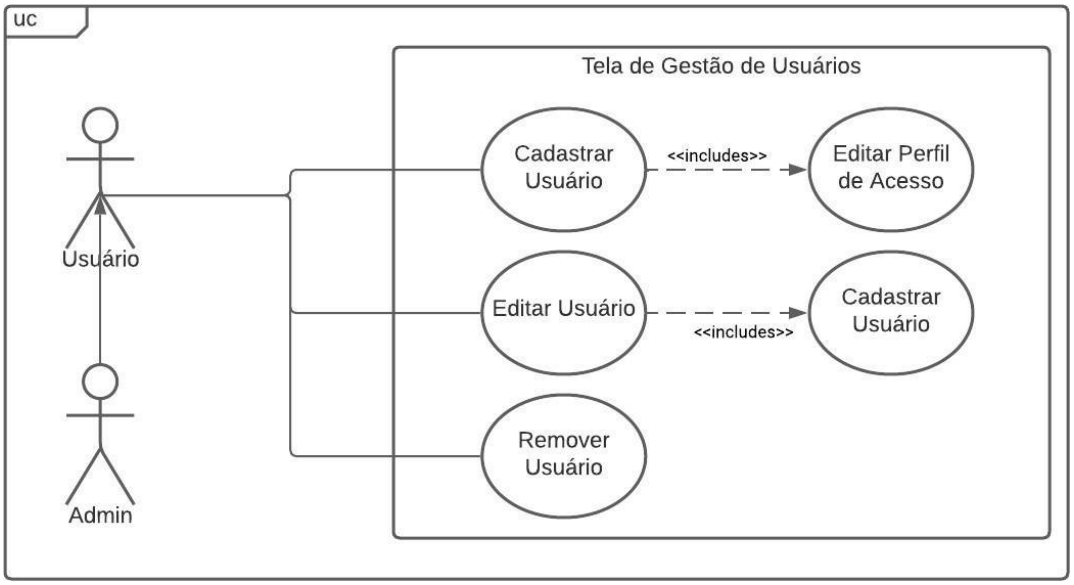

Elaborado pela autora

Figura 68 - Diagramas de Casos de Uso - Prontuário



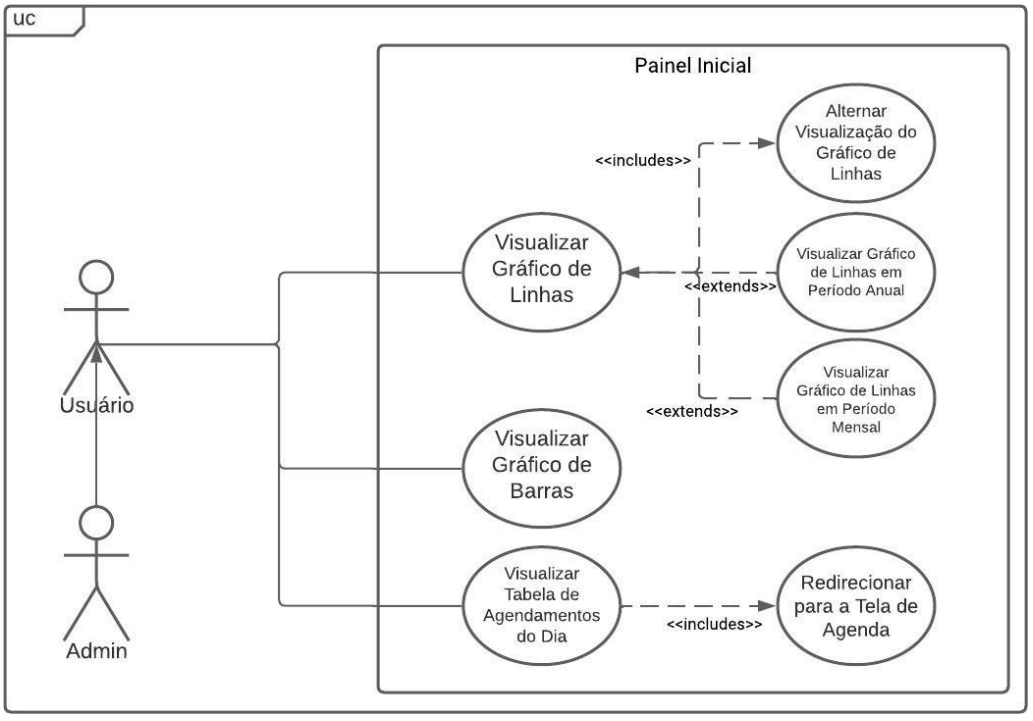

Elaborado pela autora

Figura 69 - Diagramas de Casos de Uso - Gestão de Usuários




Elaborado pela autora

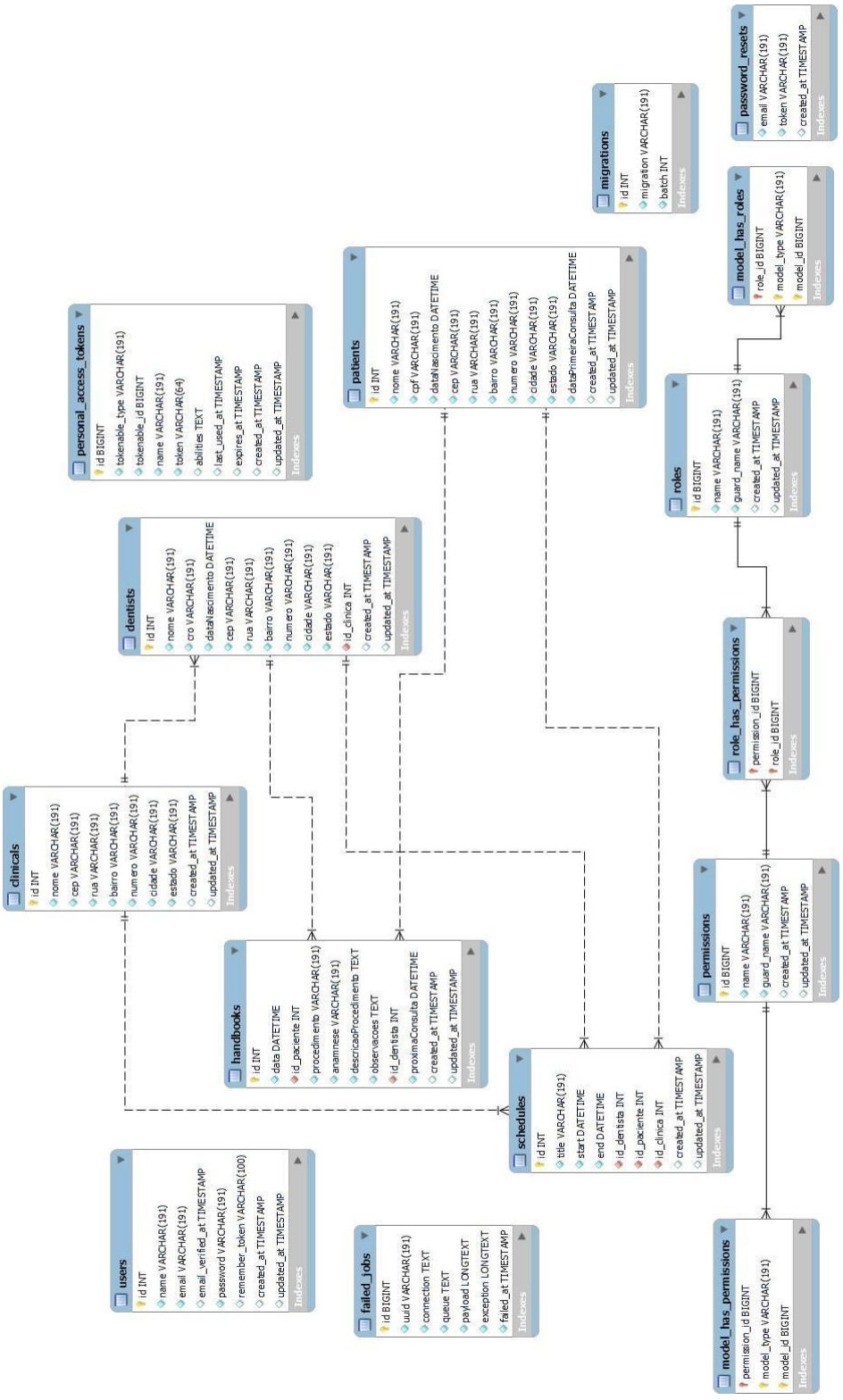
Figura 70 - Diagramas de Casos de Uso - Painel Inicial




Elaborado pela autora

# APÊNDICE B – Diagrama de entidade e relacionamento

Figura 71 - Diagrama EER – Completo




Elaborado pela autora