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

JUAN HERMANN SERAMUCIN

SISTEMA WEB PARA CONTROLE E GERENCIAMENTO DE AGENDA EM CENTROS ESTÉTICOS

TRABALHO DE CONCLUSÃO DE CURSO

PATO BRANCO
2019

JUAN HERMANN SERAMUCIN

SISTEMA WEB PARA CONTROLE E GERENCIAMENTO DE AGENDA EM CENTROS ESTÉTICOS

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

Orientadora: Prof.ª Andreia Scariot Beulke
Coorientador: Prof. João Guilherme Brasil Pichetti

PATO BRANCO
2019

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

TERMO DE APROVAÇÃO

TRABALHO DE CONCLUSÃO DE CURSO

SISTEMA WEB PARA CONTROLE E GERENCIAMENTO DE AGENDA EM CENTROS ESTÉTICOS

POR

JUAN HERMANN SERAMUCIN

Este trabalho de conclusão de curso foi apresentado no dia 09 de outubro de 2019, como requisito parcial para obtenção do título de Tecnólogo em Análise e Desenvolvimento de Sistemas, pela Universidade Tecnológica Federal do Paraná. O acadêmico foi arguido pela Banca Examinadora composta pelos professores abaixo assinados. Após deliberação, a Banca Examinadora considerou o trabalho aprovado.

Banca examinadora:

Profa. MSc
Andréia Scariot Beulke

Prof. MSc
Vinicius Pegorini

Profª Drª
Mariza Miola Dosciatti

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

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

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

RESUMO

Este projeto teve como objetivo desenvolver um sistema *web* que controle e gerencie os agendamentos oferecidos por centros estéticos. O cliente poderá agendar um serviço de forma rápida e eficiente por meio de uma agenda *on-line*, na qual ele seleciona o dia e o horário que deseja ser atendido para um ou mais serviços, podendo, ainda, escolher o profissional que vai atendê-lo. A agenda é construída dinamicamente de acordo com o horário de início e término do expediente cadastrados no sistema. Também é possível bloquear os horários em que houver indisponibilidade de atendimento. O sistema possibilita controlar os agendamentos dos clientes e dos profissionais que prestam serviços. Dentre as tecnologias utilizadas no trabalho proposto, está a linguagem Java e os *frameworks* Spring, JavaServer Faces e Primefaces.

Palavras-chave: Centros estéticos. Sistema *web.* PrimeFaces. JavaServer Faces.

ABSTRACT

This project aimed at developing a web system which controls and manages schedules offered by aesthetic centers. The cliente will be able to schedule a service quickly and efficiently using an on-line agenda in which he chooses the date and time he wants to have the service done and he will also be able to choose the corresponding professional. The agenda is built dynamically according to the starting time and business time registered in the system. It is also possible to block timetables when there is no possibility of service. The system controls both the schedules of clients and also the professionals who provide the service. Among the technologies used in the proposed work, there is Java language and the frameworks Spring, JavaServer Faces and PrimeFaces.

Keywords: Aesthetic Center. Web system. PrimeFaces. JavaServer Faces.

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE SIGLAS E ACRÔNIMOS

ABIHPEC	Associação Brasileira da Indústria de Higiene Pessoal, Perfumaria e Cosméticos
AJAX	*Asynchronous Javascript and XML*
ASP	*Active Server Pages*
CGI	Comitê Gestor da Internet
CGI	Comitê Gestor da Internet
CSS	*Cascading Style Sheet*
HTML	*HyperText Markup Language*
HTTP	*HyperText Protocol Transfer*
IoC	Inversão de Controle
JCP	Java *Community Process*
JSF	JavaServer Faces
MVC	Model, View, Controller
PHP	*HiperText PreProcessor*
RF	Requisitos Funcionais
RIA	*Rich Internet Applications*
RNF	Requisitos Não Funcionais
URL	*Uniform Resource Locator*
WWW	*World Wide Web*

SUMÁRIO

1 INTRODUÇÃO

Este capítulo apresenta as considerações iniciais, os objetivos e a justificativa da realização deste trabalho.

1.1 CONSIDERAÇÕES INICIAIS

A Internet, desde seus primórdios, vem oferecendo recursos e padrões cada vez mais eficientes para auxiliar seus usuários a desenvolverem suas tarefas e rotinas, pois permite acesso rápido e independente de plataforma.

Com a tecnologia disponível em vários segmentos da indústria e do comércio, os sistemas de informação permitem que muitos serviços sejam realizados de forma automática, além de permitir o gerenciamento e controle das informações registradas por meio desses sistemas.

É notório o aumento de usuários de Internet no Brasil proporcionado pela facilidade de acesso disponível por diversos dispositivos que oferecem esse serviço, como, computadores, celulares e TVs e pela variedade de atividades que a Internet oferece seja para uso profissional ou pessoal. Dados do Comitê Gestor da Internet (CGI) apontam que cerca de 54% das residências possuem acesso à Internet. O número de empresas que possuem *web* sites varia de acordo com o porte, sendo 87% para grandes empresas, 75% para médias empresas e 52% para pequenas empresas. Para perfis em redes sociais a pesquisa aponta que 60% dos usuários são de empresas de grande porte, 56% de médio porte e 50% de pequeno porte. Dentre as atividades mais frequentes realizadas na Internet os dados revelam que 99% dos usuários utilizam a Internet para envio de e-mails, 94% para buscar informações sobre produtos ou serviços, 88% para fazer pagamentos, transferências e consultas bancárias via *Internet banking*. (CGI, 2016).

Além dos serviços citados existem softwares ou aplicações *web* que visam oferecer a realização de atividades específicas de acordo com o segmento da empresa. No que tange o comércio de centros estéticos, por exemplo, as atividades realizadas são relacionadas ao tratamento corporal e facial de seus clientes. As alternativas de serviços a serem ofertados variam de acordo com o tipo de cliente que envolve os gêneros feminino e masculino de várias idades (SEBRAE, 2015).

De acordo com a Associação Brasileira da Indústria de Higiene Pessoal, Perfumaria e Cosméticos (ABIHPEC), o mercado de centros de estética fatura cerca de 38 bilhões no Brasil,

com crescimento de 10% ao ano e o Brasil é o terceiro maior mercado do mundo nessa área (ABIHPEC, 2015).

No que tange os salões de beleza desempenham um papel fundamental para elevar a autoestima de seus clientes. Segundo a Euromonitor International (2017), a indústria de produtos voltados para beleza e cuidados pessoais cresceu a uma taxa média ao ano de 13% entre 2004 e 2014. Em um mercado que movimenta anualmente US$30 bilhões no país, esse nível de crescimento anual adiciona US$2 bilhões dentro do mercado global de beleza.

Nesse contexto, este projeto tem como objetivo desenvolver um sistema *web* que controle e gerencie os agendamentos de dias, horários e serviços oferecidos pelos centros estéticos. O cliente poderá agendar um serviço de forma rápida e eficiente, além de se comunicar diretamente com o estabelecimento por via chat próprio.

1.2 OBJETIVOS

A seguir são apresentados o objetivo geral e específicos do sistema proposto neste trabalho. O objetivo geral está relacionado com o resultado principal da realização deste trabalho e os objetivos específicos complementam o geral em termos de funcionalidades do sistema.

1.2.1 Objetivo Geral

Desenvolver um sistema *web* para controle e gerenciamento de agenda de um centro estético.

1.2.2 Objetivos Específicos

Para alcançar o objetivo geral, serão estabelecidos os seguintes objetivos específicos.

- Permitir a centralização e automatização das informações cadastradas para o agendamento.
- Proporcionar flexibilidade, agilidade e eficiência no agendamento de clientes para realização de um serviço.
- Permitir a consulta de dados por meio de relatórios.

1.3 JUSTIFICATIVA

Para uma empresa manter-se no mercado é necessário, além de oferecer serviços e produtos de qualidade, que esteja em constante atualização seja para os produtos e serviços que oferecem ou pelo atendimento a seus clientes. Para isso, é fundamental que as empresas invistam em tecnologias, equipamentos e serviços de qualidade para que possam auxiliar no desenvolvimento de suas atividades e funções. Os sistemas de informação são ferramentas necessárias para o controle e gerenciamento dessas atividades, pois dentre suas diversas funcionalidades, está a parte de cadastros e relatórios, que são funcionalidades básicas, mas de grande importância, pois permite a centralização dos mais variados dados em um banco de dados. Além disso, existem funcionalidades específicas visando atender as necessidades de cada tipo de empresa seja no nível operacional, tático ou estratégico.

A facilidade de acesso às mais variadas informações e o atendimento a serviços de vários segmentos proporcionados pelo uso da tecnologia faz com que as pessoas necessitem cada vez mais de atendimentos rápidos e pontuais.

No que compete aos centros de estética, os clientes são pessoas de gêneros, raças e idade distintas que buscam a prestação de serviços relacionados ao tratamento estético corporal e facial. Esses centros trabalham com hora marcada para cada serviço prestado que tem um tempo limite entre o início e o fim do atendimento e são realizados por profissionais distintos de acordo com a sua especialidade. Assim, a agenda desse tipo de estabelecimento é importante para o controle e gerenciamento dos serviços oferecidos para que ocorram nos horários estabelecidos para cada cliente. Normalmente, o agendamento de um determinado serviço ocorre por uma chamada telefônica ou mensagem instantânea, sendo necessário um atendente para controlar a agenda entre os profissionais que executam os serviços e os clientes do estabelecimento.

Nesse sentido, este trabalho se justifica pela proposta de oferecer aos clientes de um centro de estética a possibilidade de realizar um agendamento *on-line*, escolhendo dia, horário, tipo de serviço e profissional que deseja para executar o serviço. Além disso, o sistema visa centralizar os dados dos agendamentos em um único banco de dados o que permite que a empresa possa realizar consultas a diversas informações, como, por exemplo, serviços realizados por profissional, serviços mais realizados, entre outros.

Dentre as principais tecnologias utilizadas no desenvolvimento do trabalho proposto, está a linguagem Java e o *framework* Spring e as tecnologias JavaServer Faces e Primefaces. O Spring é baseado nos padrões de Inversão de Controle (IoC) e Injeção de Dependências

(SOUZA, 2015). O Spring Security permite deixar o sistema mais seguro, pois garante a autenticação e autorização dos usuários, permitindo que somente quem é cadastrado no sistema tenha acesso aos recursos restritos. Além disso, é importante que o sistema seja responsivo, pois o acesso poderá ocorrer por diversos dispositivos, como *tablets*, celulares e computadores e, assim, é importante manter o layout e acesso às finalidades adequados para cada dispositivo utilizado para o acesso.

1.4 ESTRUTURA DO TRABALHO

Este texto está organizado em capítulos e este é o primeiro que apresenta a introdução, as considerações iniciais, os objetivos e a justificava de realização deste trabalho. O Capítulo 2 apresenta o referencial teórico. No Capítulo 3 são apresentados os materiais e o método utilizados para o desenvolvimento do trabalho. No Capítulo 4 está o resultado da realização do trabalho que apresenta a modelagem do sistema, a apresentação e a implementação do sistema. A conclusão é apresentada no Capítulo 5 e, por fim, estão as referências citadas no texto.

2 APLICAÇÕES WEB

Este capítulo apresenta a fundamentação teórica deste trabalho relacionado as tecnologias utilizadas para desenvolvimento do trabalho. Estas tecnologias que estão relacionadas ao conceito de aplicações ricas e *Asynchronous Javascript and XML* (AJAX).

2.1 DESENVOLVIMENTO WEB TRADICIONAL

As aplicações *web* possuem características específicas com relação a sistemas convencionais, pois em uma aplicação *web*, além dos aspectos de engenharia definidos nas aplicações tradicionais, envolvem características específicas, como o ambiente baseado em navegação, por exemplo. Além disso, é importante não confundir uma aplicação *web* com um *website*, pois uma aplicação *web* enfatiza aspectos relacionados à aplicabilidade e funcionalidade, enquanto os *websites* têm ênfase na apresentação, estruturação e navegação.

Em uma aplicação *web* tudo é processado por um servidor que recebe uma requisição do cliente, ou seja, as aplicações *web* são feitas por requisição e resposta, no qual o cliente (usuário) faz uma requisição para um servidor e o servidor responder para o cliente. O *browser* permite ao usuário solicitar um recurso e o servidor pode responder com vários recursos, conforme a solicitação, como: uma página *HyperText Markup Language* (HTML), uma figura ou documentos que serão apresentas ao usuário. Os conteúdos dessas páginas são estilizados com recursos do *Cascading Style Sheet* (CSS) e do JavaScript para comandos interativos.

A comunicação entre o cliente e servidor é realizada por meio de um protocolo, chamado *HyperText Protocol Transfer* (HTTP) que é baseado em requisições e respostas. Uma requisição possui um método HTTP, a página que será acessada e os dados do formulário. A resposta possui um status que informa se a solicitação foi realizada, o tipo e o conteúdo.

Cada requisição o cliente especifica o nome de um método a ser aplicado em um recurso no servidor e o *Uniform Resource Locator* (URL) desse recurso. Os métodos HTTP mais conhecidos são o *get* e o *post*. No método *get* o URL é dado como argumento, no qual o servidor responderá retornando os dados identificados por este URL. O método *post* especifica o URL de um recurso que pode tratar os dados fornecidos no corpo do pedido.

Conforme destaca Fraternali, Rossi e Sánchez-Figueroa (2010) a *World Wide Web* (WWW) foi uma plataforma para acessar recursos estáticos e dinâmicos codificados em HTML e que a interação do usuário era limitada a navegar pelas páginas por meio dos *links* e inserir dados em formulários sendo páginas *web* eram simples e limitadas. O autor enfatiza, ainda, que

as primeiras tentativas de estender a funcionalidade das interfaces (como *applets* em Java e scripts do lado cliente) enriqueceram a navegação baseada em HTML, com objetos interativos, efeitos de animações e validações de entrada. No entanto, a utilização desses novos recursos causou problemas de padronização e problemas de arquitetura.

A partir disso, o desenvolvimento de aplicações *web* passaram por transformações, pois houve um progresso no aumento de funcionalidades de tarefas realizadas na *web* abrindo espaço para soluções *web* mais modernas que se assemelham a aplicações *desktop* e permitem interações sofisticadas com o usuário, processamento do lado cliente, comunicação assíncrona e multimídia.

Assim, as tecnologias evoluíram e sugiram as *Rich Internet Applications* (RIAs) que desempenham um importante papel no desenvolvimento de interfaces *web*.

2.2 DESENVOLVIMENTO COM INTERFACE RICA

Conforme destaca Fraternali, Rossi e Sánchez-Figueroa (2010) o termo RIA refere-se a um conjunto heterogêneo de soluções que permitem adicionar novas capacidades à *web* tradicional baseada em hipertexto. Conforme destaca Meliá *et al.* (2010) oferecem melhor capacidade de resposta e uma experiência para o usuário mais extensa que as aplicações *web* tradicionais.

As *Rich Internet Application* (RIA), surgiram em 1995, mais especificamente com a criação dos *applets*. As RIAs proporcionam a criação de aplicações *web* que permitem aos navegadores uma maior riqueza de recursos ao invés de utilizar somente HTML tradicional da época. O uso de *applets* era a alternativa encontrada por muitos desenvolvedores quando necessitavam criar aplicações *web* mais complexas. Ao longo dos anos, o HTML e o JavaScript evoluíram, solucionando algumas limitações e problemas existentes nas versões iniciais. A grande evolução, entretanto, veio por meio do surgimento do AJAX que é conjunto de técnicas para desenvolvimento *web* focadas em JavasScript dotadas de recursos que proporcionam uma interação mais rica e rápida para o usuário (PINA; OLIVEIRA, 2013).

Fraternali, Rossi e Sánchez-Figueroa (2010) afirmam que as RIAs são aplicações cliente-servidor e que são utilizadas no desenvolvimento de aplicações desktop e *web* e fornecem benefícios de implantação e manutenção dos aplicativos da *web*, ao mesmo tempo em que suportam uma Interface do Usuário (UI) do cliente muito mais rica. Além disso, as RIAs introduzem novas características arquitetônicas no campo das aplicações tradicionais da

Web, e uma UI com estados conectados e desconectados, uma comunicação cliente e servidor inteligente com solicitações assíncronas.

Como citado por Vinícius Baggio Fuentes (2014, p. 74) “depois da repopularização do JavaScript com o uso intenso do AJAX, os browsers têm se tornado muito mais poderosos que antigamente”. E é nesse requisito que se refere uma interface rica, na qual não existe a necessidade de atualização de todas as páginas, recurso que era possível somente com o uso do HTML. As RIAs trabalham com o conceito do AJAX no qual não existe a necessidade de atualizar a página para atualizar alguma informação que contenha nela.

Graças à RIA ter sua “Cliente Engine” existem vários benefícios ao se criar uma interface rica, como, uma melhor resposta entre servidor/cliente passando a impressão ao usuário utilizar um programa *desktop*, uma comunicação assíncrona fazendo com que quando o usuário clique em uma opção sem a necessidade de esperar a resposta do servidor.

2.3 COMUNICAÇÃO SÍNCRONA E ASSÍNCRONA

No desenvolvimento de aplicações *web* há dois tipos de requisições a serem feitas para um servidor: a síncrona e a assíncrona. Ao utilizar requisição síncrona o browser fica parado aguardando o término da requisição, o que contradiz todo o propósito do AJAX.

Na comunicação síncrona os processos de origem e destino são sincronizados a cada mensagem, o *browser* ficará parado até a requisição for completada. Esse procedimento acontece da seguinte forma (COLOURIS *et al.*, 2013 p. 147): quando um envio é realizado, o processo de origem é bloqueado até que a recepção correspondente seja realizada. Assim, quando uma recepção é realizada, o processo é bloqueado enquanto a mensagem não chegar e a transmissão da mensagem ocorre paralelamente com o processo de origem. Colouris *et al.*, (2013, p. 148) explicam, também, que na comunicação assíncrona a operação não bloqueia o processo, ou seja, o processo origem pode prosseguir assim que a mensagem seja copiada para um buffer local e a transmissão da mensagem ocorre em paralelo com o processo de origem.

Um exemplo de requisição assíncrona é o e-mail do Google, o Gmail. Por meio da técnica de *pooling* ele cria constantemente requisições para verificar a chegada de novos *e-mails* no servidor. Dessa forma, enquanto o e-mail não chega, o usuário pode interagir com outras funcionalidades do sistema de *e-mails* (MILETTO, 2014).

3 MATERIAIS E MÉTODO

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

3.1 MATERIAIS

O Quadro 1 apresenta a listagem das principais ferramentas e tecnologias que serão utilizadas no desenvolvimento do sistema proposto.

Quadro 1 - Lista de ferramentas e tecnologias utilizadas

Ferramenta / Tecnologia	Versão	Disponível em	Aplicação
Astah Community	7.2.0	http://astah.net/editions/community	Modelagem do sistema: desenvolvimento do diagrama de classes e de casos de uso
Bootstrap	4	https://getbootstrap.com/docs/3.3/	*Framework front-end*
Jaspersoft Studio	6.4.3	https://community.jaspersoft.com/download	Gerador de relatórios.
Java	8	http://www.java.com/pt_BR/download/win10.jsp	Linguagem de programação
JavaServer Faces	2.3	https://javaee.github.io/javaserverfaces-spec/	Especificação Java para a construção de interfaces
Lombok	1.18.8	https://projectlombok.org/	Biblioteca Java focada em produtividade e redução de código redundante.
pgAdmin 4	1.22.23.0	https://www.pgadmin.org/download/	Ferramenta para administração do banco de dados.
PostgreSQL	9.610	https://www.postgresql.org/	Banco de dados
PrimeFaces	6.2	https://www.primefaces.org/downloads/	*Framework front-end*
Spring Boot	1.5.8	https://projects.spring.io/spring-boot/	*Framework* para criação de aplicações
Spring Data	2.0.0	https://projects.spring.io/spring-data-jpa/	*Framework* de implementação de repositórios
Spring Security	4.2.3	https://projects.spring.io/spring-security/	*Framework* de autenticação e segurança da aplicação.
Spring Tools Suite	4.38.2	https://spring.io/tools	Ambiente de desenvolvimento para o sistema

Fonte: Autoria própria.

2.2 JAVASERVER FACES

Com a popularização do desenvolvimento de sistemas web, muitas tecnologias ganharam notoriedade, como *HiperText PreProcessor* (PHP) e *Active Server Pages* (ASP) e, posteriormente Java (CORDEIRO, 2014). A infraestrutura para o desenvolvimento web é baseada em *servlets* que são classes desenvolvidas em Java que ficam no servidor e são invocadas com algum padrão URL.

JSF é uma tecnologia que foi definida pelo *Java Community Process* (JCP) que cria padrões e especificações de tecnologias desenvolvidas em Java. Essa tecnologia é baseada no padrão *Model, View, Controller* (MVC) e, portanto, separa o desenvolvimento do sistema em camadas distintas (*model*o, visão e controlador). Cada camada tem a sua responsabilidade. A camada *model* é responsável por representar os objetos do negócio, manter o estado da aplicação e fornecer ao controlador o acesso aos dados. Na camada *view* ficam as interfaces gráficas da aplicação e é responsável em definir a forma como os dados são apresentados e encaminhar as ações dos usuários o controlador e, por fim, a camada *controller* trata qualquer requisição integrando a *model* e a *view*, ou seja, pega a entrada do usuário e envia para a *model*, recebe a resposta e retorna para a *view* (FOWLER, 2003).

O JavaServer Faces (JSF) utiliza um modelo de interfaces gráficas baseado em eventos e por basear-se no padrão MVC, uma de suas principais vantagens é a separação entre a visualização e as regras de negócio (GEARY, HORSTAMANN, 2007). O JSF permite a interação com dispositivos clientes e fornece recursos para que o desenvolvimento da aplicação visual, lógica da aplicação e lógica de negócios de uma aplicação *web*.

Segundo Silva e Foschini (2012), os serviços mais importantes do JSF, são: arquitetura MVC: divide a estrutura da aplicação em camadas; conversão de dados: facilita a tarefa de converter os dados e customizar as regras de conversão desses dados; validação e manipulação de erros: auxilia na criação de regras de validação para campos obrigatórios; internacionalização: fornece suporte à internacionalização, como a codificação de caracteres e a seleção de pacotes de recursos; componentes customizados: permite a criação de componentes personalizados e a utilização de componentes de terceiros e; renderizadores alternativos: produz tipos de saídas diferentes do padrão.

A especificação do JSF define seis fases distintas. Essas fases são (GORLA; FOSCHINI, 2013):

- Restaurar a visão: constrói ou recupera a árvore de componentes que é a estrutura de dados com objetos Java para todos os elementos de interface gráfica.

- Aplicar valores de requisição: realiza interações sobre os objetos dos componentes e verifica quais valores requisitados pertencem ao objeto em questão e armazena-os.
- Processar validações: converte a *string* de valores armazenada para valores locais. Se os dados são válidos o ciclo de vida do JSF continua e, caso ocorram erros de conversão ou validação, o JSF chama a fase de renderizar resposta e exibe a página atual para que o usuário insira novamente os dados.
- Atualizar valores do *model*o: atualiza os valores do *model*o depois de serem validados e convertidos.
- Invocar aplicação: um método é executado quando o botão ou link for disparado. Esse método retorna uma *string* de resultado que é aplicada às regras de navegação e, após isso, o usuário é direcionado para a página adequada de acordo com o resultado da execução da ação.
- Renderizar resposta: renderiza a página de resposta e envia para o navegador.

Para fazer a sincronização entre os componentes de interface de usuário com os objetos Java da camada *model* da aplicação são utilizados *Managed Beans* que atuam do lado servidor da aplicação do JSF (SILVA; FOSCHINI, 2012). Os *Managed Beans* são responsáveis por fornecer dados que serão exibidos nas telas, receber os dados enviados nas requisições e executar as tarefas de acordo com as ações dos usuários (LIMA, MAGALHÃES, 2014).

De acordo com Lima e Magalhães (2014) um *Managed Bean* é definido pela criação de duas classes Java, sendo, a primeira registrada no arquivo faces-config.xml e, a segunda, criada com a anotação *@ManagedBean* do pacote *javax.faces.bean*. Com essa anotação o JSF assume por padrão que o nome do *Managed Bean* é o nome da classe.

Uma forma de implementar uma interface gráfica *web*, que possa ser denominada como rica, é utilizando o *PrimeFaces* e é utilizado em aplicações *web* com a tecnologia JSF.

2.3 PRIMEFACES

PrimeFaces é uma biblioteca de componentes ricos em JSF e que inclui diversos componentes que são necessários para o desenvolvimento de interface gráfica para a *web*.

Segundo Coelho (2012), as características do *PrimeFaces* são: biblioteca de componentes para JSF de código aberto; possui um rico conjunto de componentes visuais; suporte ao AJAX nativo; possui temas pré-definidos e oferece a possibilidade de criação de temas; possui documentação de fácil acesso e exemplifica o uso dos componentes.

Alguns recursos do *PrimeFaces* são (COELHO, 2012):

- *Datatable* com seleção com um click: permite que com um click seja exibido o valor de uma linha selecionada;
- *Drag and drop*: arrastar e soltar os objetos na interface;
- Notificador: envia uma resposta para o usuário indicando se a ação foi realizada com sucesso.
- *Auto complete*: completa a informação que o usuário informa em um campo de entrada;
- *Poll*: realiza chamadas a um *Managed Bean* sem a necessidade de ser disparada por um usuário.

3.2 MÉTODO

O método utilizado neste trabalho foi o processo sequencial linear de Pressman (2008) que define as fases de levantamento de requisitos, análise e projeto e desenvolvimento do sistema.

Na fase inicial do levantamento de requisitos foi realizada entrevista com um profissional da área de estética para poder compreender as necessidades de uma empresa que atua nessa área. Dentre os requisitos levantados, observou-se a necessidade de ter um sistema que controlasse, mais especificamente, a agenda de um centro de estética, na qual o cliente pudesse ter mais autonomia e flexibilidade para agendar seus próprios horários, sem a necessidade de um atendente. Além disso, o cliente poderia escolher por qual profissional desejaria ser atendido.

Na fase de análise e projeto do sistema tendo como base os requisitos preestabelecidos, definiram-se os casos de uso do sistema que foram documentados gerando informações para a modelagem do banco de dados.

Após a modelagem dos casos de uso, foi elaborado o diagrama de entidades e relacionamentos do banco gerando as tabelas com seus campos, tipos, tamanhos e relacionamentos.

Na fase do desenvolvimento foram implementadas as telas e o código-fonte do sistema. Os testes foram realizados informalmente, visando obter possíveis erros nas funcionalidades e no código-fonte, a medida em que o sistema era desenvolvido.

4 RESULTADOS

Este capítulo apresenta o resultado deste trabalho que é o desenvolvimento de um sistema *web* para controle e gerenciamento de agenda de centros estéticos.

4.1 ESCOPO DO SISTEMA

O sistema proposto é para a *web* e permite controlar e gerenciar a agenda de um centro de estética. Os clientes poderão realizar o agendamento de um serviço *on-line*, escolhendo dia, horário, tipo de serviço e profissional desejado para executar o serviço.

O sistema tem quatro atores, sendo: o administrador que terá acesso a todas as funcionalidades do sistema, o profissional que terá acesso às funcionalidades relacionadas à sua agenda, o atendente que terá acesso às funcionalidades relacionadas aos agendamentos e bloqueios e o cliente que poderá realizar seu cadastro para autenticação para, posteriormente, realizar seus agendamentos.

As funcionalidades do sistema em relação aos cadastros envolvem o de usuários, serviço e da agenda. Os clientes poderão realizar o cadastro de um agendamento. Para isso, ele deverá selecionar os serviços desejados, e se houver mais de um profissional cadastrado no sistema, aparecerá uma caixa para a seleção. Para a seleção da data será disponibilizado um *input* com uma interface em formato de calendário. E por último terá um c*omboBox* com todos os horários disponíveis.

Para que o cliente possa realizar o agendamento, o sistema deve permitir os cadastros de serviços e dos profissionais que atendem no estabelecimento e vinculá-los a um determinado serviço.

Para fazer um agendamento o cliente deverá registrar-se e, posteriormente, autenticar-se no sistema. Assim, poderá agendar seu horário por meio da seleção de um ou mais serviços, de um profissional, do dia e do horário para o serviço ser realizado. Ao realizar o cadastro, o cliente pode realizar somente um agendamento e o administrador será responsável por aceitar ou não o cadastro do cliente e, após validado o cadastro, o cliente poderá realizar outros agendamentos.

Quando o cliente realizar um agendamento por outro meio que não seja o sistema, o atendente deverá registrá-lo no sistema para manter o controle e gerenciamento da agenda.

O sistema também permitirá que o administrador possa emitir relatórios de clientes que mais utilizaram os serviços do estabelecimento, dos serviços mais realizados, entre outros.

4.2 MODELAGEM DO SISTEMA

Este capítulo apresenta a modelagem, a apresentação e a implementação do sistema proposto nesse trabalho.

4.2.1 Requisitos Funcionais

O Quadro 2 apresenta os requisitos funcionais (RF) identificados para o sistema.

Quadro 2 - Requisitos funcionais do sistema

Identificação	Nome	Descrição
RF01	Manter agenda	Manutenção dos dados relacionados aos agendamentos.
RF02	Manter usuários	Manutenção dos dados relacionados aos usuários do sistema.
RF03	Manter serviço	Manutenção dos tipos de serviços que podem ser limpeza de pele e depilação, por exemplo.
RF04	Bloquear agenda	Permitir ao usuário administrador bloquear ou editar os horários na agenda.
RF05	Validar cadastro de novos clientes	Permitir ao usuário administrador aceitar ou rejeitar o cadastro de novos usuários.
RF06	Emitir relatórios	Proporcionar relatórios para o administrador sobre as atividades do estabelecimento como, por exemplo, atividades prestadas por funcionário, serviços mais procurados, horários mais requisitados, entrada mensal, serviços realizados por funcionário.
RF07	Atribuir serviços	O administrador atribui os serviços que o profissional pode realizar.
RF08	Atribuir permissões	O administrador pode atribuir permissões para outro usuário. Por exemplo: quando o cadastro é realizado o administrador deve validar o cadastro e, posteriormente, torná-lo um profissional.
RF09	Desativar usuário	O administrador poderá inativar um usuário cadastrado.
RF10	Manter empresa	Manutenção dos dados relacionados aos dados da empresa.

Fonte: Autoria própria.

4.2.2 Requisitos Não Funcionais

O Quadro 3 exibe os requisitos não funcionais (RNF) identificados para o sistema.

Quadro 3 – Requisitos não funcionais do sistema

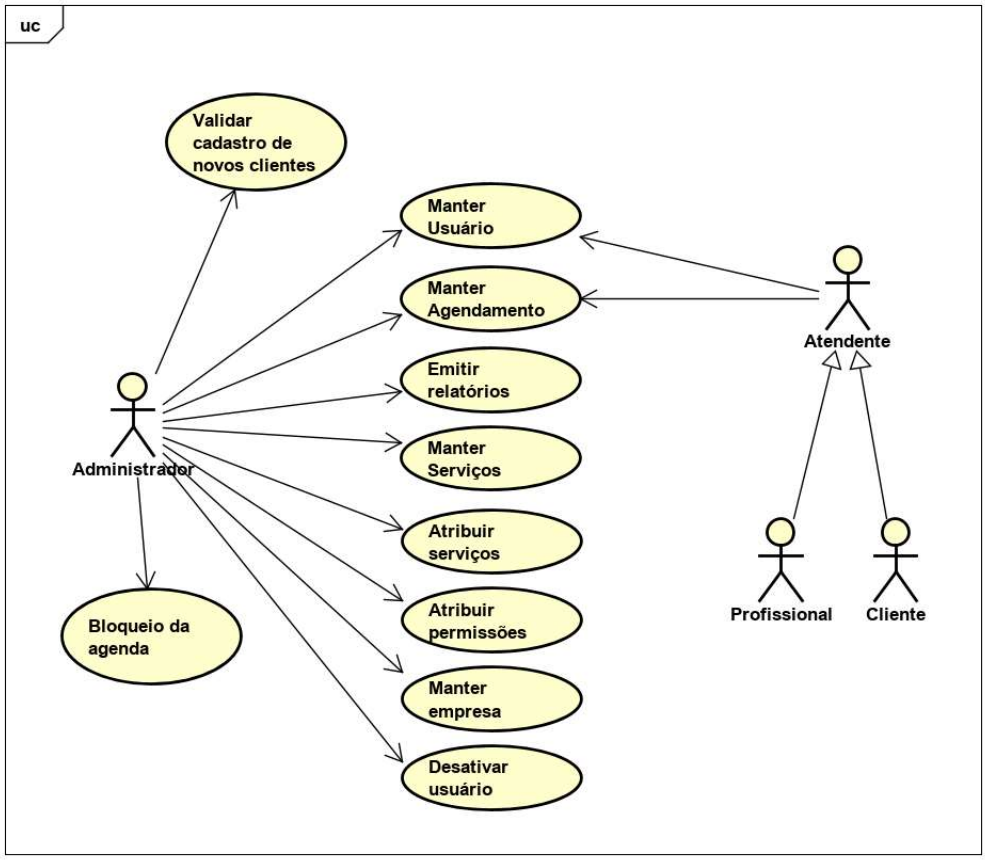
Identificação	Nome	Descrição
RNF01	Efetuar autenticação	O sistema validará o *login* e senha informados pelo usuário para acessar o sistema.

RNF02	Campos de preenchimento obrigatórios	Os campos que são de preenchimento obrigatório serão validados por meio de uma função do sistema.
RNF03	Campos com máscaras de entrada	Os campos que possuem caracteres especiais serão validados por meio de máscaras de entrada.
RNF04	Vínculo	Dados relacionados devem estar vinculados. Por exemplo, um serviço deve estar relacionado a um profissional. O sistema também não deve permitir a exclusão de dados vinculados (não é possível excluir um profissional se possuir serviços relacionados).
RNF05	Relatórios	A data inicial deve ser maior que a data final.
RNF06	Remover agendamento	O cliente não poderá remover um agendamento com período inferior a 24 horas para ser realizado.
RNF07	Selecionar serviços	As opções de serviço serão mostradas conforme o profissional que foi selecionado.
RNF08	Controle de horário na agenda	Só poderá ser mantido na agenda se o profissional selecionado estiver livre no horário selecionado.

Fonte: Autoria própria.

A Figura 1 apresenta os casos de uso definidos para o sistema que é composto pelo administrador, atendente, profissional e cliente. O administrador tem acesso a todas as funcionalidades do sistema. O atendente, o profissional e o cliente tem acesso ao manter usuário, agendamentos.

Figura 1 – Casos de uso



Fonte: Autoria própria.

No Quadro 4 é descrito a operação do tipo “manter” que se refere à inclusão dos registros de usuários, agendamentos e tipos de serviço. O funcionamento da inclusão de um novo registro é padrão para todos os casos de uso relacionados a esse tipo de operação.

Quadro 4 – Operação “incluir” dos casos de uso de cadastro

Caso de uso: Inserir registros. **Descrição:** Ator inclui dados no sistema. **Atores:** Administrador, Atendente, Profissional e Cliente. **Pré-condição:** Não há. **Sequência de Eventos:** 1. Ator acessa a tela para cadastrar as informações solicitadas. 2. O sistema insere as informações no banco de dados. **Pós-Condição:** Registro inserido no banco de dados.

Nome do fluxo alternativo (extensão)	**Descrição**
1. Campos obrigatórios não informados.	1.1. Ator deixa de informar dados obrigatórios e clica em salvar. 1.2. O sistema valida que não foram informados todos os campos obrigatórios e exibe mensagem ao usuário sem salvar o registro. 1.3. O sistema permanece na tela de inclusão mantendo os dados informados anteriormente.

Fonte: Autoria própria.

O Quadro 5 apresenta o funcionamento da operação de alterar registro de todos os casos de usos identificados como “manter”.

Quadro 5 – Operação “alterar” dos casos de uso de cadastro

Caso de uso:
Alterar registro.
Descrição:
Ator altera dados no sistema.
Atores:
Administrador, Atendente, Profissional e Cliente.
Pré-condição:
Dados cadastrados no sistema.
Sequência de Eventos:
1. O ator acessa a tela para visualização de dados já cadastrados.
2. O sistema apresenta o registro selecionado para a alteração.
3. Ator altera os dados do registro e clica em salvar.
4. O sistema valida as informações e salva as informações no mesmo registro.

Pós-Condição:
Registro alterado no banco de dados.

Nome do fluxo alternativo (extensão)	**Descrição**
1. Campos obrigatórios não informados.	1.1. Ator apaga dados obrigatórios e clica em salvar. 1.2. O sistema valida que não foram informados todos os campos obrigatórios e exibe mensagem ao usuário sem salvar o registro. 1.3. O sistema permanece na tela de edição mantendo as alterações realizadas.
2. Campos informados em formato incorreto.	2.1. Ator altera os dados deixando em um formato incorreto e clica em salvar. 2.2. O sistema valida que os dados não estão no formato esperado e exibe mensagem ao usuário sem salvar o registro. 2.3. O sistema permanece na tela de edição mantendo as alterações realizadas.

O Quadro 6 apresenta o funcionamento da operação de exclusão dos casos de uso do tipo “manter”.

Quadro 6 - Operação “excluir” dos casos de uso de cadastro

Caso de uso: Excluir registro. **Descrição:** Ator solicita a exclusão de dados no sistema. **Atores:** Administrador, Atendente, Profissional e Cliente. **Pré-condição:** Dados cadastrados no sistema. **Sequência de Eventos:** 1. O ator acessa a tela para visualização de dados já cadastrados. 2. O sistema apresenta o registro selecionado para a exclusão. 3. Ator clica em excluir registro. 4. O sistema exclui as informações do banco de dados e exibe as informações do status do procedimento. **Pós-Condição:** Registro excluído no banco de dados.	
Nome do fluxo alternativo (extensão)	**Descrição**
Cliente deseja excluir consulta	1.1 O cliente não poderá excluir um agendamento com período inferior a 24 horas para o serviço ser realizado.
1. Exclusão de registro com vínculos no sistema	1.1. Ator clica em excluir um registro que possui vínculos no sistema. 1.2. O sistema verifica que o registro tem vínculos, não o exclui e exibe mensagem de alerta ao usuário.

Fonte: Autoria própria.

O Quadro 7 apresenta o funcionamento da operação de consultar um registro dos casos de uso do tipo “manter”

Quadro 7 – Operação “consultar” dos casos de uso de cadastro

Caso de uso: Consultar registro. **Descrição:** Ator solicita a consulta de dados cadastrados no sistema. **Atores:** Administrador, Atendente, Profissional e Cliente. **Pré-condição:** Dados cadastrados no sistema. **Sequência de Eventos:** 1. Ator acessa a tela para visualização de dados já cadastrados. 2. O sistema exibe os dados da consulta ao usuário.

Pós-Condição: Dados são exibidos aos usuários.

Fonte: Autoria própria.

O Quadro 8 apresenta o caso de uso referente aos relatórios do sistema.

Quadro 8 – Caso de uso dos relatórios do sistema

Caso de uso: Visualizar relatórios. **Descrição:** Visualizar os relatórios de atividades prestadas por funcionário, serviços mais prestados, horários mais requisitados, valor de entrada (lucro) e funcionários. **Ator:** Administrador. **Pré-condição:** Relatório gerado/visualizado. **Sequência de Eventos:** 1. O ator acessa a tela de relatórios; 2. O ator escolhe os parâmetros de filtragem para o relatório; 3. O sistema gera o arquivo e/ou a página de impressão. **Pós-Condição:** Relatório gerado e visualizado.	
Nome do fluxo alternativo (extensão)	**Descrição**
1. Não existem dados referentes ao relatório	1.1. O sistema informa ao ator que não foi possível gerar o relatório. 1.2. Retorna ao fluxo principal do caso de uso.
2. Data incorreta	2.1 Ator seleciona data final menor que a data inicial. 2.2 Sistema exibe mensagem informando que o erro ocorrido.

Fonte: Autoria própria.

O Quadro 9 apresenta a expansão do caso de uso para bloquear a agenda.

Quadro 9 – Caso de uso de bloqueio do sistema

Caso de uso: Bloquear agenda **Descrição:** Ator deseja bloquear algum horário na agenda do estabelecimento. **Atores:** Administrador. **Pré-condição:** Não há. **Sequência de Eventos:** 1. O ator acessa a tela de bloqueio da agenda; 2. O ator escolhe o(s) dia(s) e/ou a(s) hora(s) que deseja bloquear a agenda.

Pós-Condição:
Agenda bloqueia o tempo escolhido.

Fonte: Autoria própria.

Quadro 10 - Caso de uso validação de cliente do sistema

Caso de uso:
Validar cadastro de novos clientes
Descrição:
Ator deseja aceitar ou não novos clientes cadastrados.
Atores:
Administrador.
Pré-condição:
Novos clientes adicionados
Sequência de Eventos:
1. O ator acessa a tela de novos clientes;
2. O ator escolhe um cliente;
3. O ator verifica suas informações do cadastro;
4. O ator aceita ou não o cliente cadastrado.
Pós-Condição:
Novos clientes aceitos ou não.

Fonte: Autoria própria.

Quadro 11 - Caso de uso de atribuir serviços

Caso de uso:
Atribuir serviços
Descrição:
Ator deseja adicionar ou remover serviços aos profissionais cadastrados.
Atores:
Administrador.
Pré-condição:
Profissional e serviço cadastrado no sistema.
Sequência de Eventos:
1. O ator acessa a tela de dados do profissional desejado;
2. O ator escolhe os serviços se serão feitos pelo profissional;
4. O ator salva o cadastro
Pós-Condição:
Novos serviços poderão ser prestados por determinado profissional.

Fonte: Autoria própria.

Quadro 12 - Caso de uso de atribuir permissões

Caso de uso:
Atribuir permissões
Descrição:
Ator deseja adicionar ou remover permissões aos usuários aceitos.
Atores:
Administrador.
Pré-condição:
Usuários cadastrados no sistema.
Sequência de Eventos:
1. O ator acessa a tela de dados do usuário desejado;
2. O ator seleciona as permissões que deseja dar ao usuário;

4. O ator salva o cadastro **Pós-Condição:** Novas permissões ao usuário.

Fonte: Autoria própria.

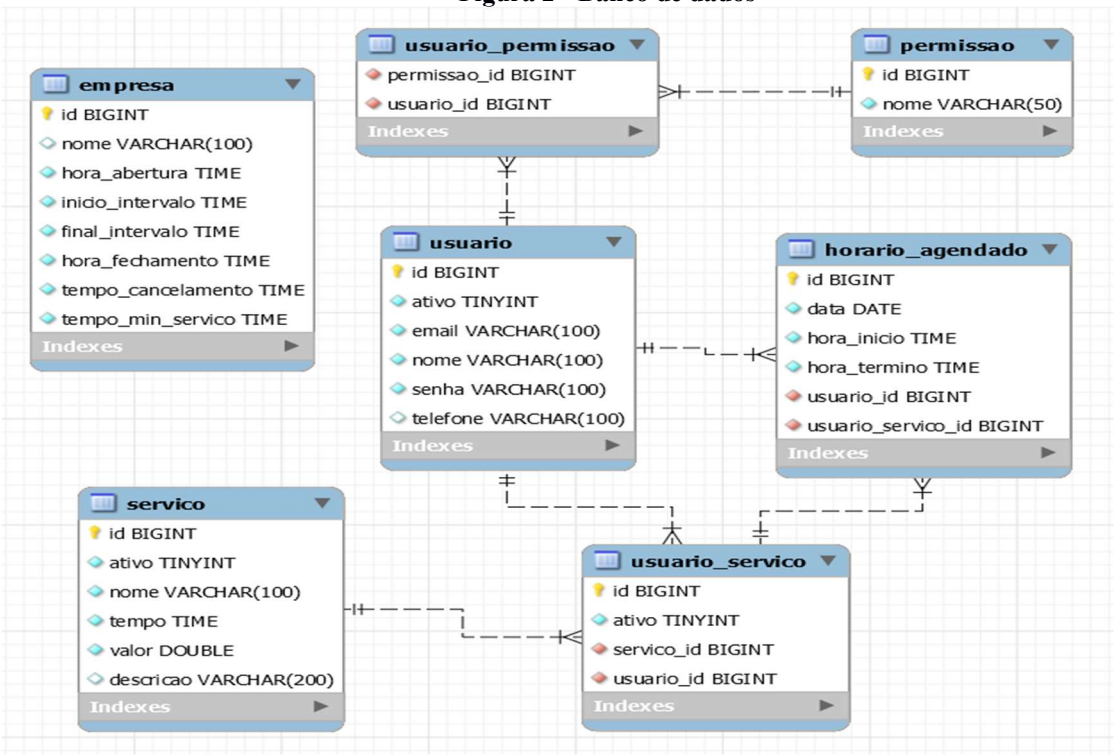
Quadro 13 - Caso de uso de desativar usuários

Caso de uso: Desativar usuário **Descrição:** Ator deseja desativar cadastro. **Atores:** Administrador. **Pré-condição:** Usuários cadastrados no sistema. **Sequência de Eventos:** 1. O ator acessa a tela de clientes ou funcionários; 2. O ator seleciona os usuários que deseja desativar; 4. O ator clica no botão de desativar usuários. **Pós-Condição:** Usuários sem permissões e desativados.

Fonte: Autoria própria.

A Figura 2 apresenta o diagrama de entidade e relacionamentos do banco de dados.

Figura 2 - Banco de dados



Fonte: Autoria própria.

A tabela de Usuario é utilizada para armazenar os dados dos usuários e suas permissões. A tabela Serviço contém os dados do serviço a ser prestado. A tabela de HorarioAgendado

contempla os dados do serviço e o profissional a ser realizado a partir da tabela Usuario_Servico, além de receber os dados do cliente pela tabela Usuario.

4.3 APRESENTAÇÃO DO SISTEMA

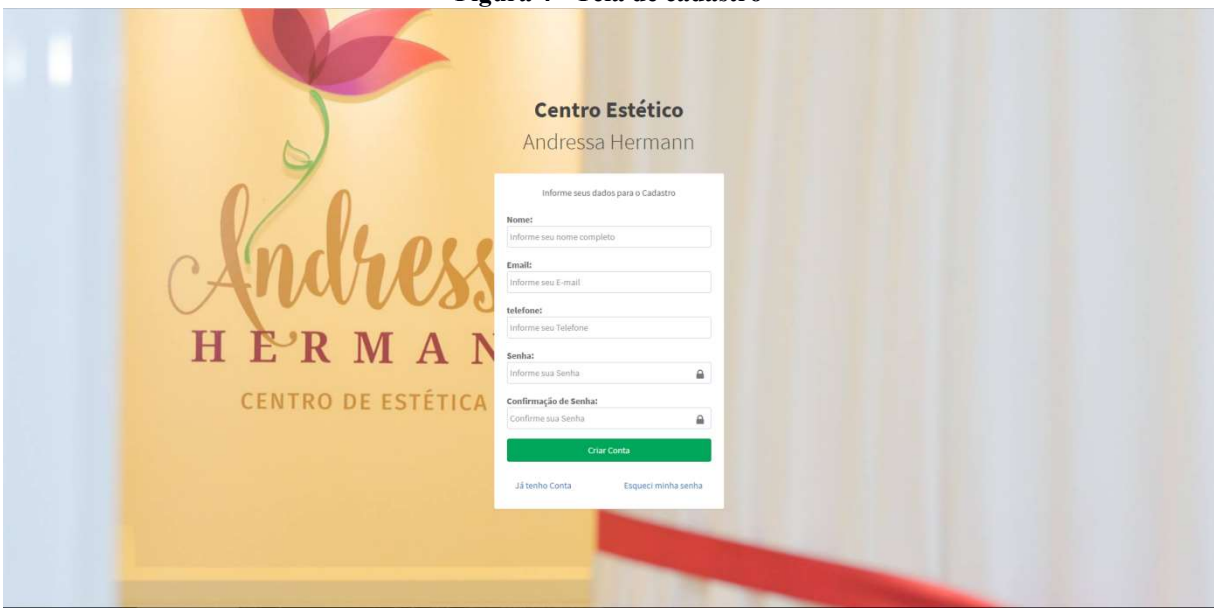
A Figura 3 representa a tela inicial, na qual o usuário deverá realizar a autenticação para ter acesso às funcionalidades do sistema, informando os dados do e-mail e senha.

Figura 3 – Tela de autenticação

Fonte: Autoria própria.

O usuário poderá realizar seu cadastro na tela apresentada na Figura 4.

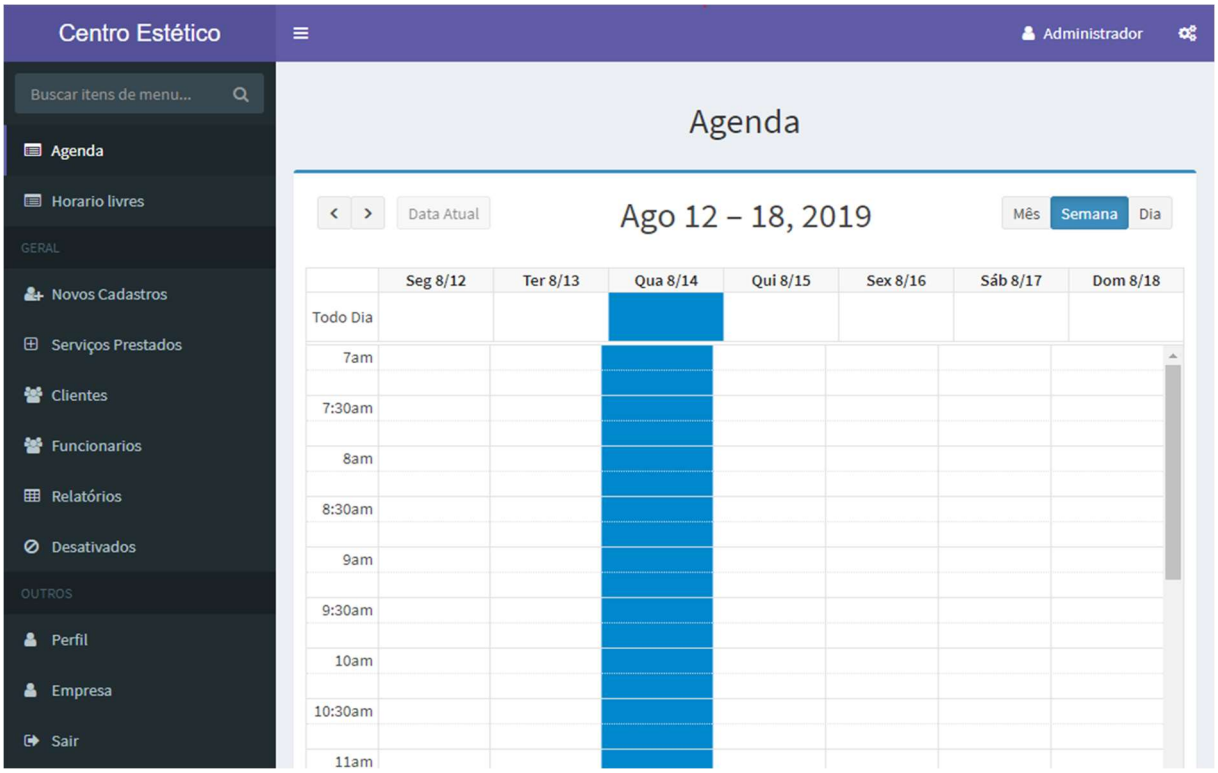
Figura 4 - Tela de cadastro



Fonte: Autoria própria.

A tela apresentada na Figura 5 é a da agenda do administrador, atendente e profissional. É por meio dessa tela que são listados os serviços cadastrados. Esses usuários poderão visualizar esses serviços filtrados por dia, semana ou mês. Por padrão, os serviços são listados por semana.

Figura 5 - Tela principal administrador, profissional e atendente



Fonte: autoria própria

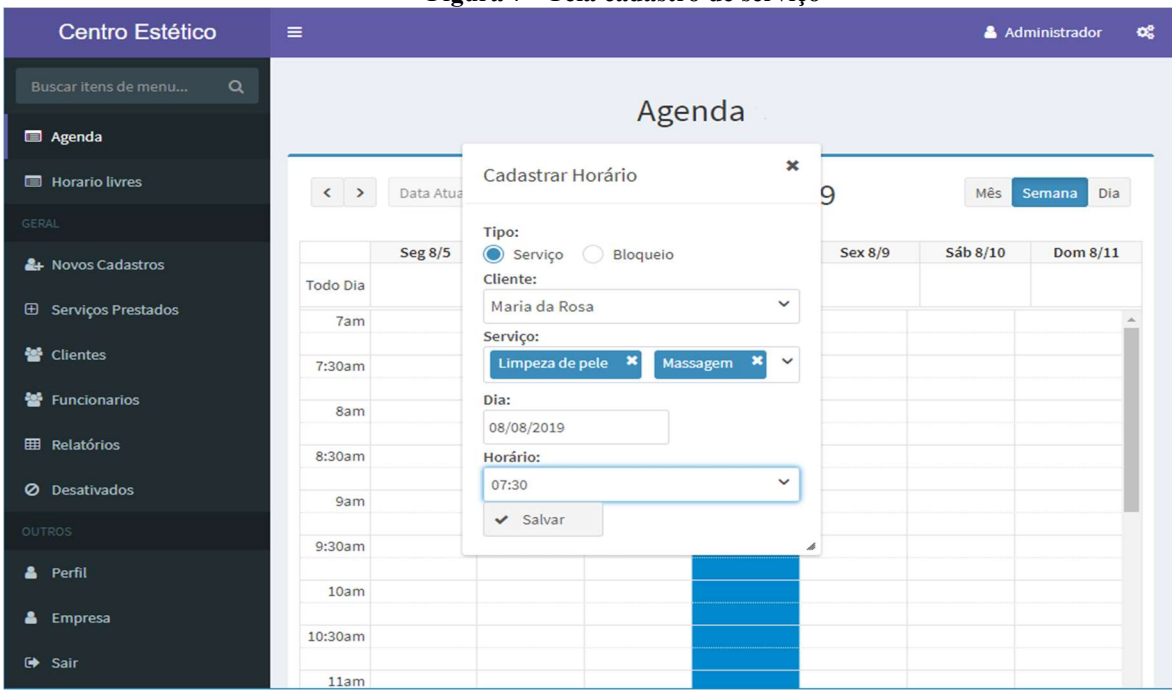
Caso haja mais de um profissional cadastrado, na tela do administrador e do atendente, aparecerá um componente *comboBox* que permite ao usuário escolher o profissional desejado para que o sistema exiba os seus serviços agendados. Também é possível exibir os serviços de todos os profissionais na mesma agenda, como mostra na Figura 6.

Figura 6 - tela principal com mais de um profissional.

Fonte: autoria própria

Ao clicar em determinado dia da agenda, o sistema abrirá uma janela *modal* para cadastrar um serviço ou realizar o bloqueio da agenda em dia e horário específicos, conforme apresentado no Figura 7.

Figura 7 - Tela cadastro de serviço



Fonte: Autoria própria.

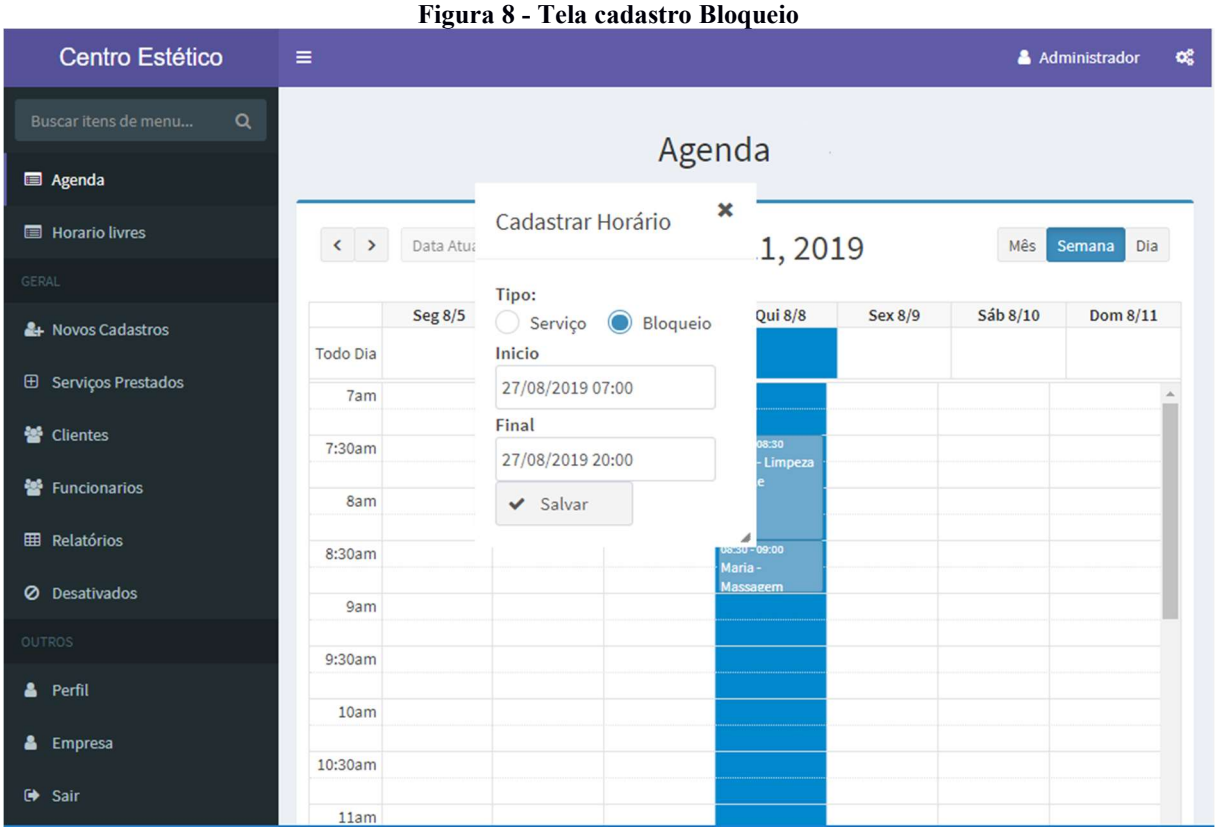
No tipo de serviço deve ser selecionado o cliente para o qual será registrado o agendamento, os serviços que serão realizados (pode ser mais de um), o dia e o horário que será realizado o serviço, conforme apresentado na Figura 7. Quando é selecionado mais de um serviço, o sistema salva cada serviço separadamente, sendo que o primeiro serviço selecionado

vai ser cadastrado com a data de início selecionada e o segundo serviço é cadastrado com a hora de início sendo a hora de término do primeiro serviço.

No tipo de bloqueio o administrador deve informar a data de início e fim do bloqueio da agenda. Por padrão esses campos são preenchidos com o horário de início e término dos atendimentos, conforme apresentado na Figura 8.

Tanto no agendamento de um serviço ou de um bloqueio, se houver mais de um profissional cadastrado, aparecerá um componente *comboBox* com a opção da escolha do profissional.

Figura 8 - Tela cadastro Bloqueio

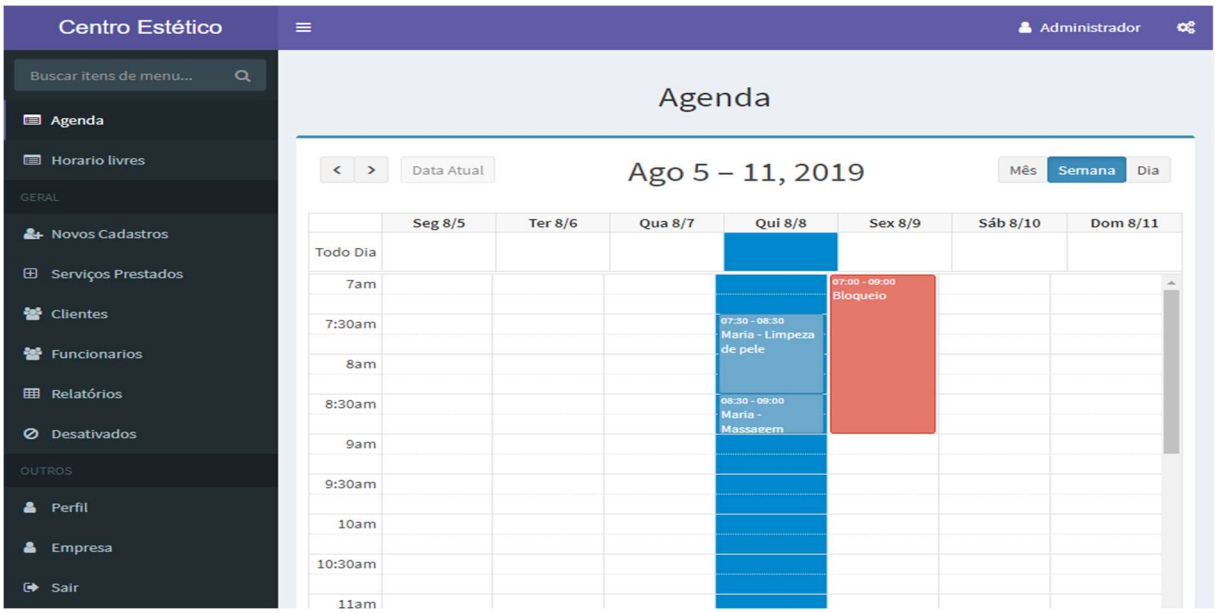


Fonte: Autoria própria.

A Figura 9 apresenta a agenda atualizada após realizar o registro do serviço ou do bloqueio em dias e horários específicos. O dia e o horário registrado para um serviço é exibido em cor azul e um bloqueio é exibido na cor vermelha. Também, caso tenha mais de um profissional cadastrado no sistema, aparecerá o nome do profissional que realizará o serviço, para caso o atendente ou administrador esteja visualizando a agenda de todos os profissionais, possa distinguir de quem é cada serviço.

Além disso, é possível mudar os dias e horários que contenham atendimentos agendados e aumentar ou diminuir o tempo do serviço por meio do conceito de *drag and drop* que permite arrastar objetos de um lado e soltá-los em outro.

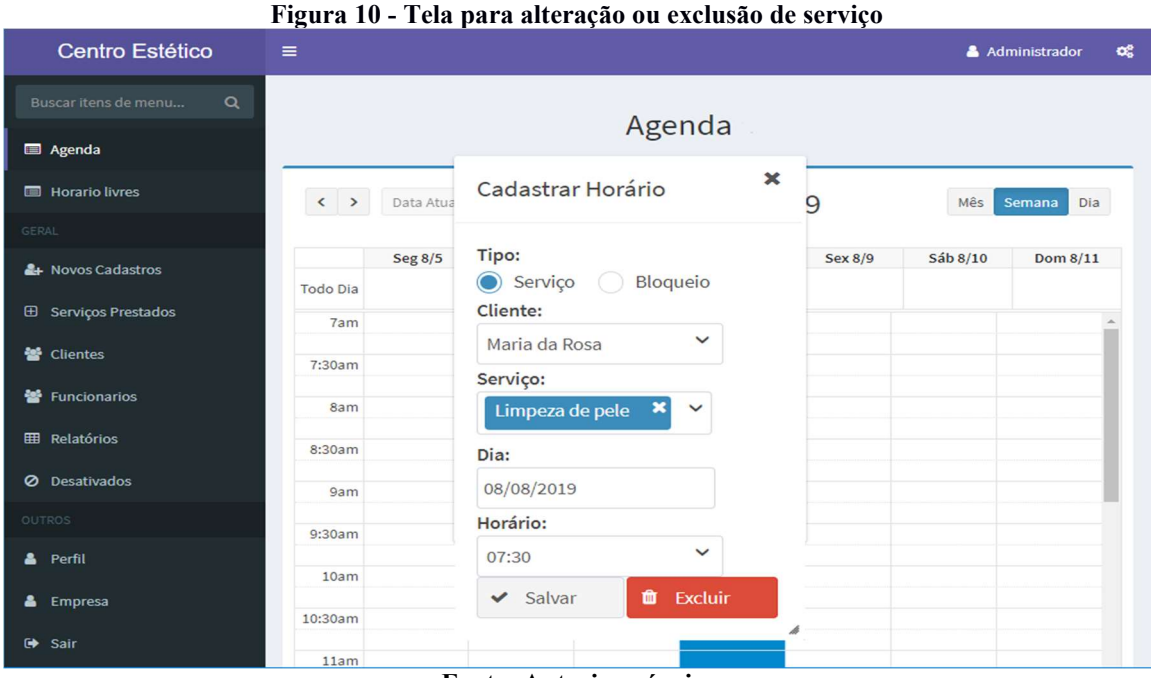
Figura 9 - Tela principal com dados cadastrados



Fonte: Autoria própria.

A Figura 10, exibe a tela para editar ou excluir um registro. Essa tela é exibida quando o usuário clicar em um serviço ou bloqueio e os dados são automaticamente preenchidos nos campos para que seja possível realizar a edição ou exclusão do registro. Caso seja clicado na área bloqueada, será exibida a tela com os dados do bloqueio preenchidos.

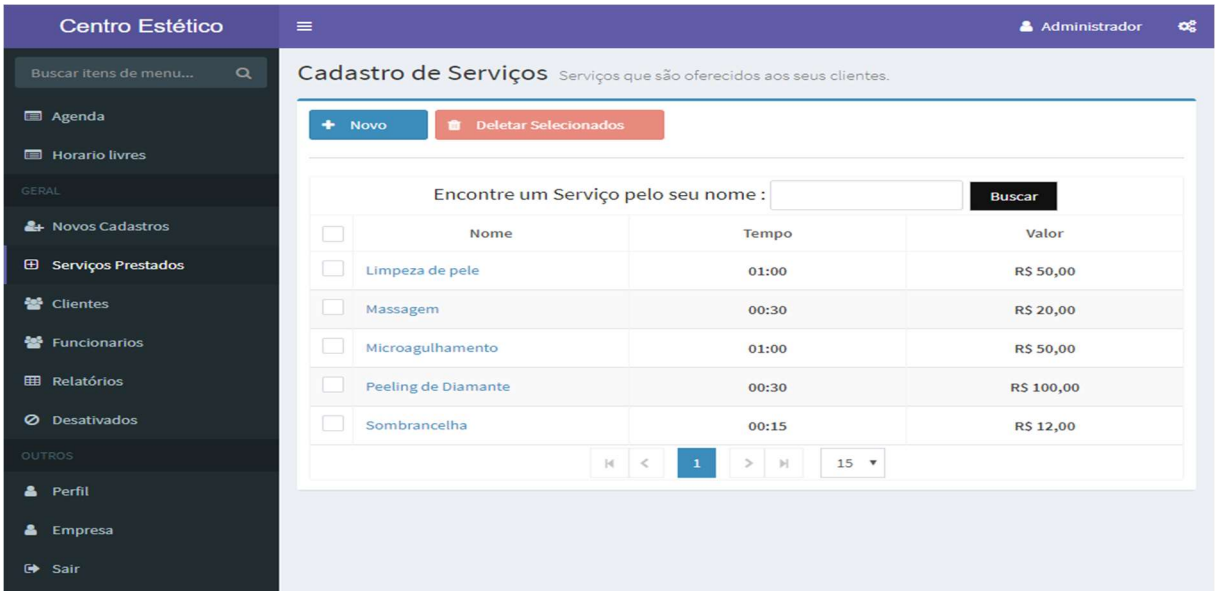
Figura 10 - Tela para alteração ou exclusão de serviço



Fonte: Autoria própria.

Na tela da Figura 11, os serviços cadastrados são exibidos em um componente *dataTable*. Nessa tela, é possível excluir registros selecionados e realizar uma busca informando o nome do serviço que deseja pesquisar. Também é possível clicar no nome do serviço para abrir a página com os dados, ou clicar no botão novo para cadastrar um novo serviço.

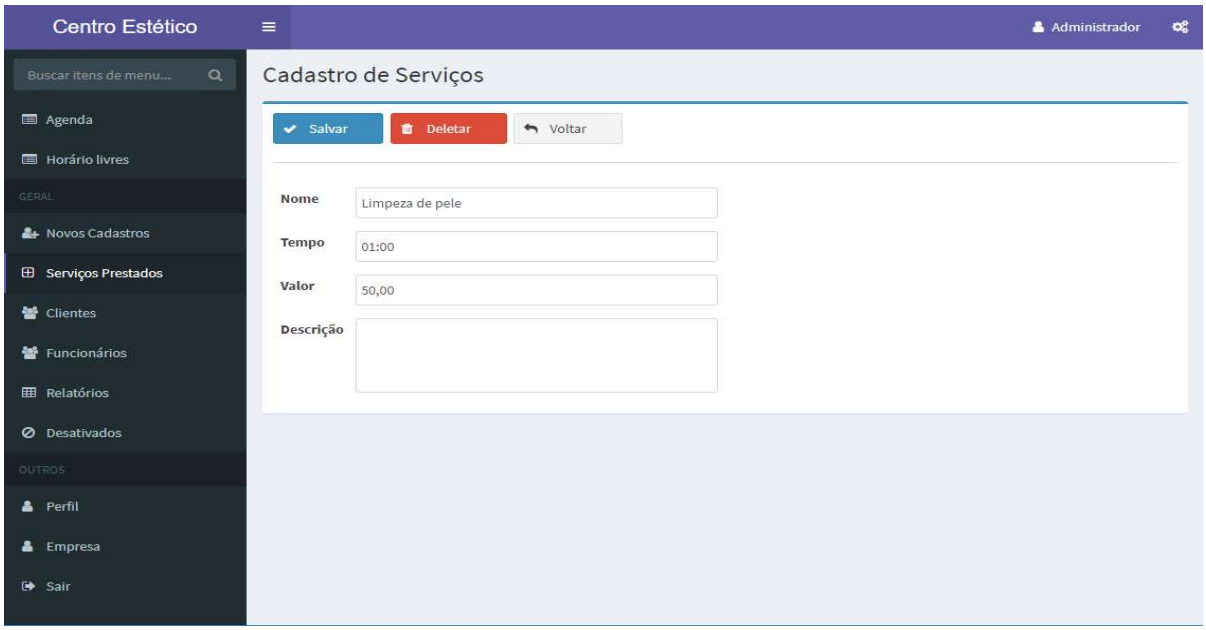
Figura 11 - Listagem de serviços



Fonte: Autoria própria.

Ao clicar no nome do serviço o sistema abrirá a tela apresentada na Figura 12, na qual é possível que um serviço seja excluído ou editado.

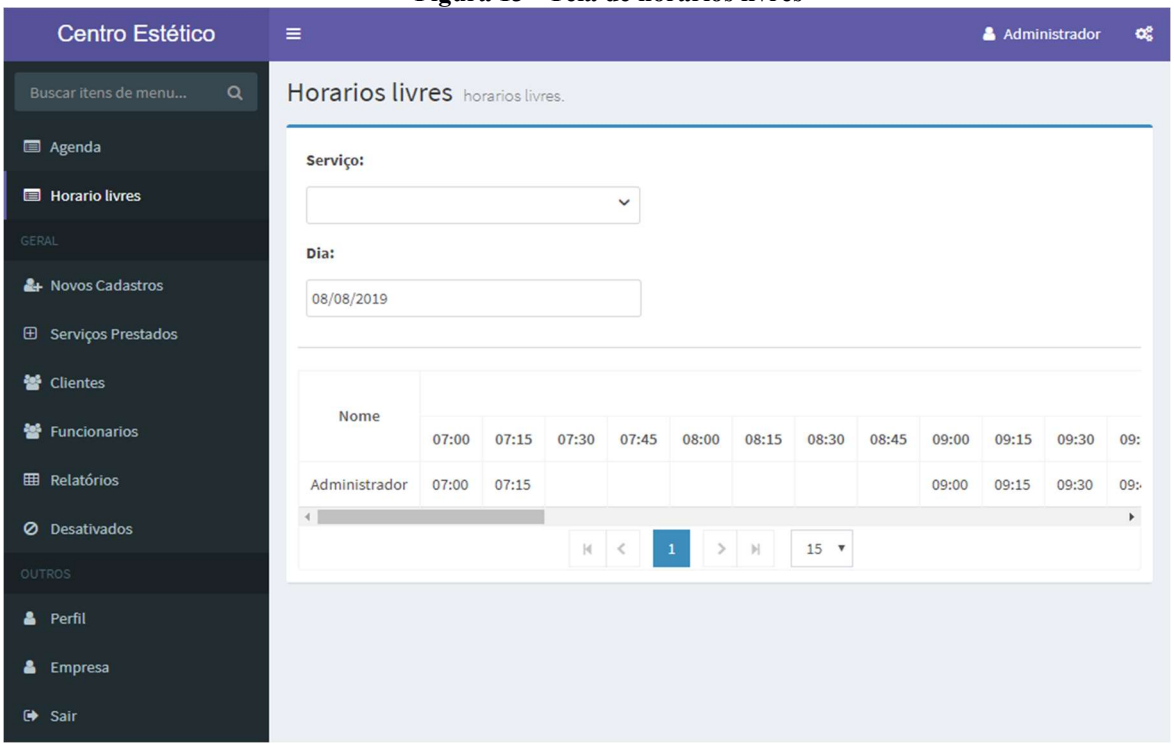
Figura 12 - Tela cadastro de serviços



Fonte: Autoria própria.

Caso o administrador ou atendente tenham que realizar uma busca dos horários que estejam livres (sem agendamentos) entre todos os profissionais cadastrados, eles podem utilizar as funções da tela apresentada na Figura 13. Nessa tela eles podem pesquisar por todos os horários que estejam livres ou fazer um filtro pelo serviço que o cliente deseja realizar e, assim, aparecerá somente os horários que estão disponíveis.

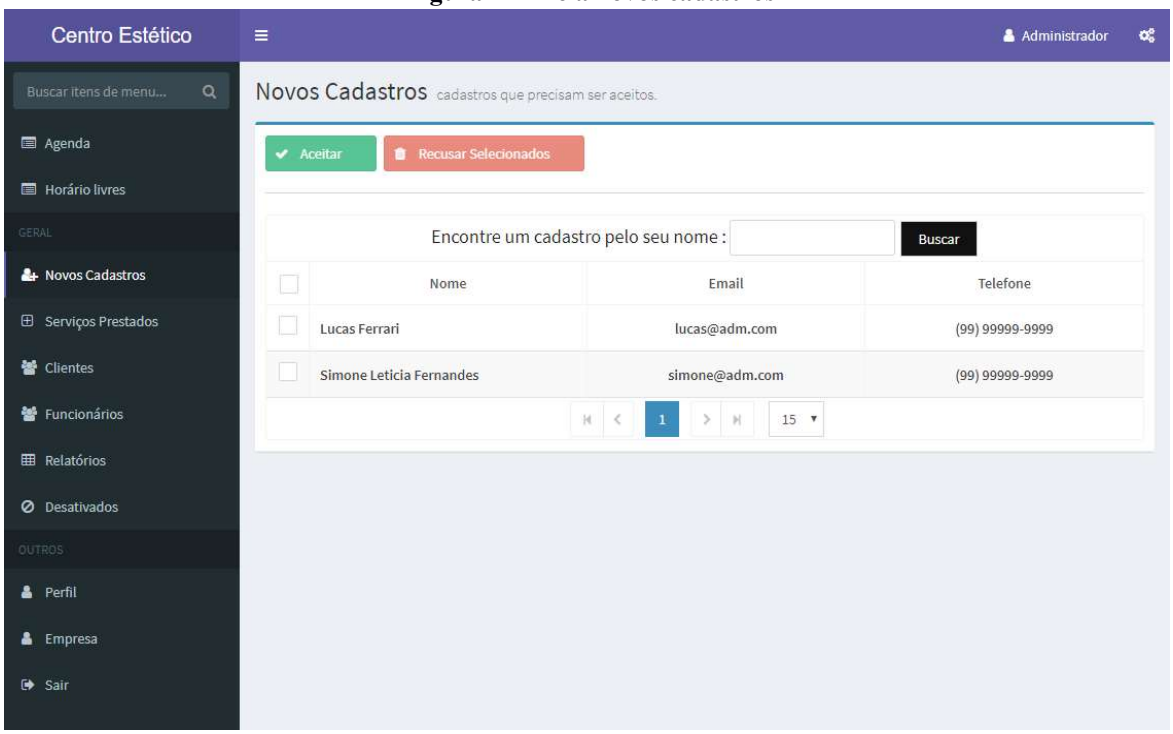
Figura 13 - Tela de horários livres



Fonte: Autoria própria.

Na tela apresentada na Figura 14 é possível visualizar todos os cadastros que foram criados e que ainda não foram aceitos. O administrador poderá aceitar ou recusar os cadastros clicando no componente *checkbox* e, posteriormente, nos botões correspondentes à ação desejada (aceite ou recuso).

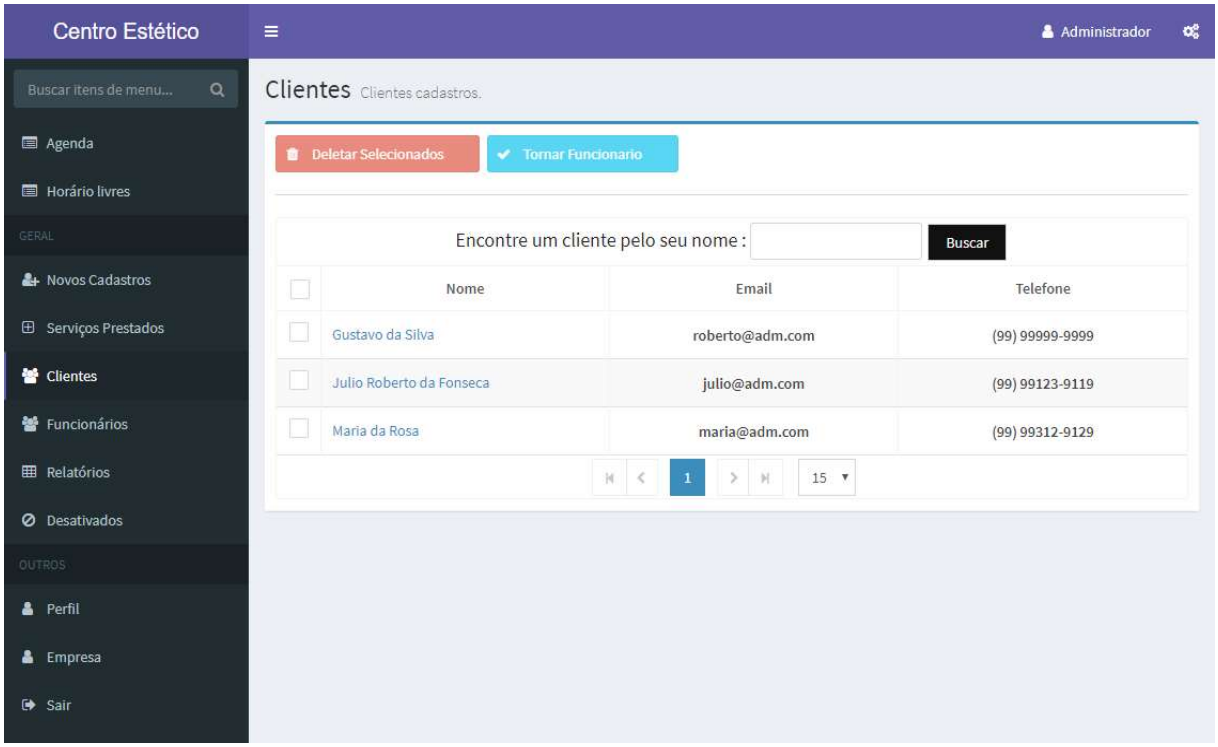
Figura 14 - Tela novos cadastros



Fonte: Autoria própria.

Na tela apresentada na Figura 15 o administrador pode gerenciar os clientes que foram aceitos e tornar um cliente em um funcionário.

Figura 15 - Tela clientes



Fonte: Autoria própria.

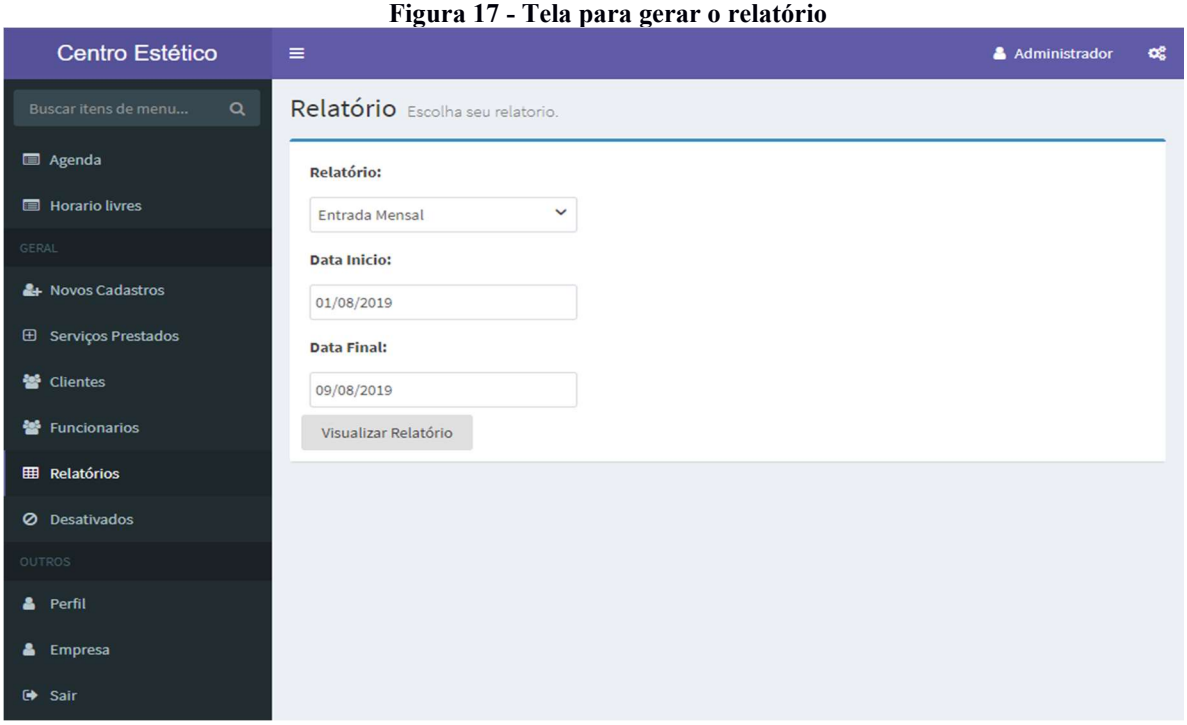
Ao clicar na opção “funcionários” do menu, o administrador tem acesso a lista de funcionários. Ao clicar no nome do funcionário serão exibidos os dados do funcionário, sendo possível alterar as informações dos serviços que o funcionário pode realizar ou alterar as permissões, conforme a Figura 16.

Figura 16 - Tela dados do funcionário

Fonte: Autoria própria.

Na opção de “relatórios” o administrador pode escolher o relatório e a data de início e término que deseja exibir os dados, conforme ilustrado na Figura 17.

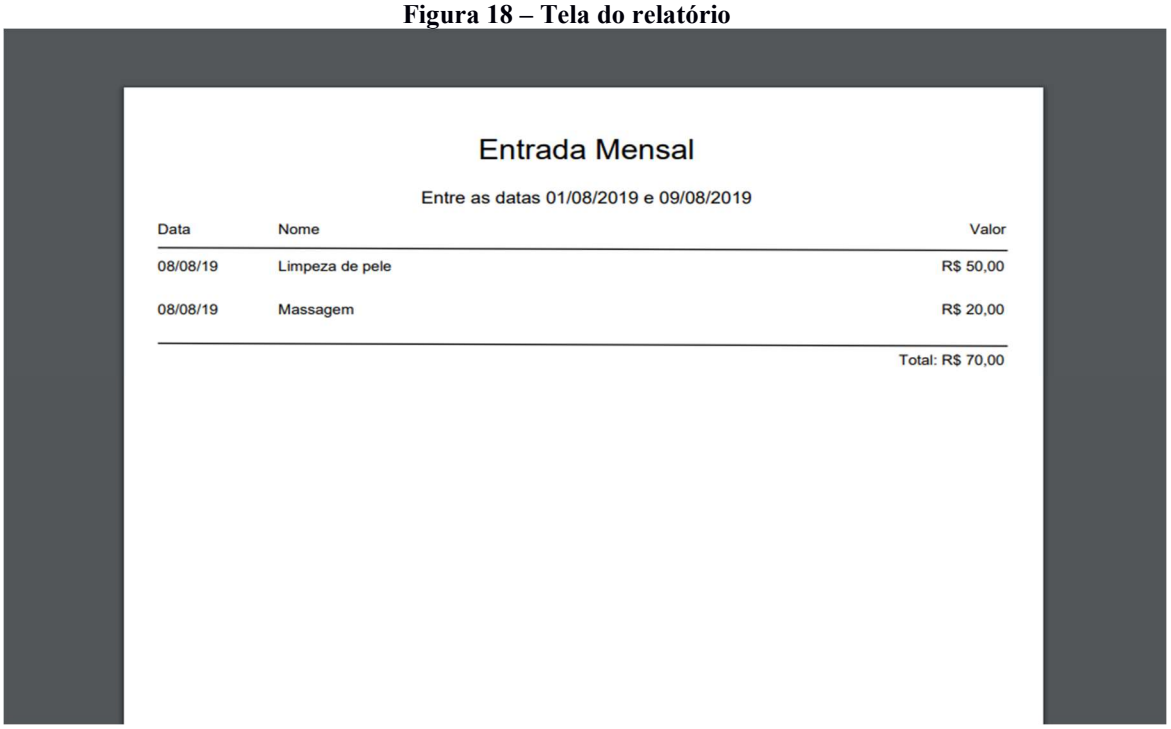
Figura 17 - Tela para gerar o relatório



Fonte: Autoria própria.

Ao clicar no botão “visualizar relatório” será aberta uma nova aba no navegador para com os dados do relatório, conforme mostra a Figura 18.

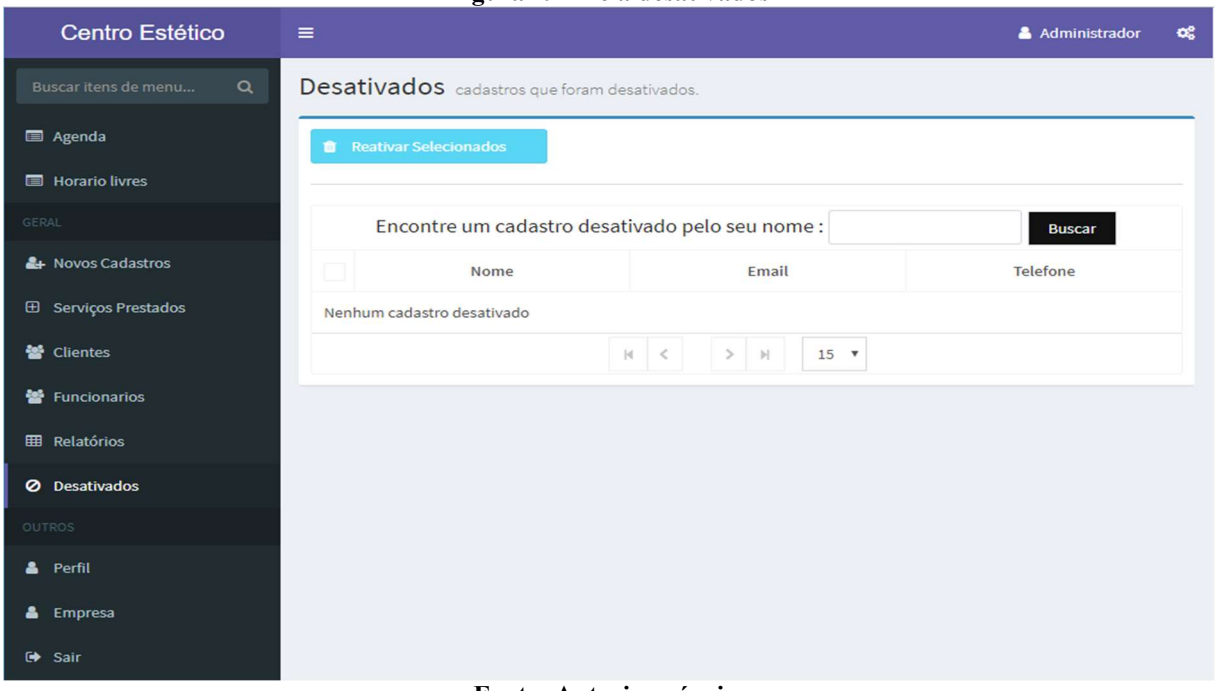
Figura 18 – Tela do relatório



Fonte: Autoria própria.

A opção “Desativados” do menu exibe todos os cadastros que foram desativados, sendo possível ativá-los novamente, caso haja necessidade, conforme apresentado na Figura 19.

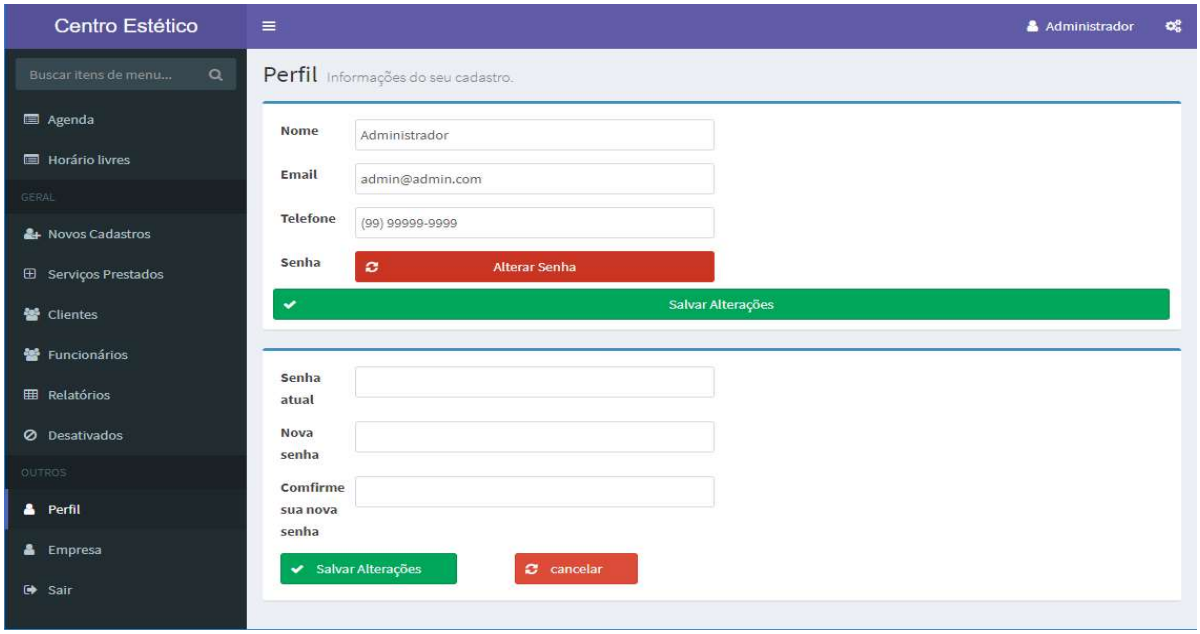
Figura 19 - Tela desativados



Fonte: Autoria própria.

Na opção “Perfil”, o usuário pode alterar as informações do seu cadastro. Ao clicar no botão para alterar a senha, será aberta a tela da Figura 20. A tela é dividida em duas partes, sendo que a primeira exibe as informações pessoais e um botão para alterar a senha e a segunda parte da tela exibe os campos para que a senha seja alterada.

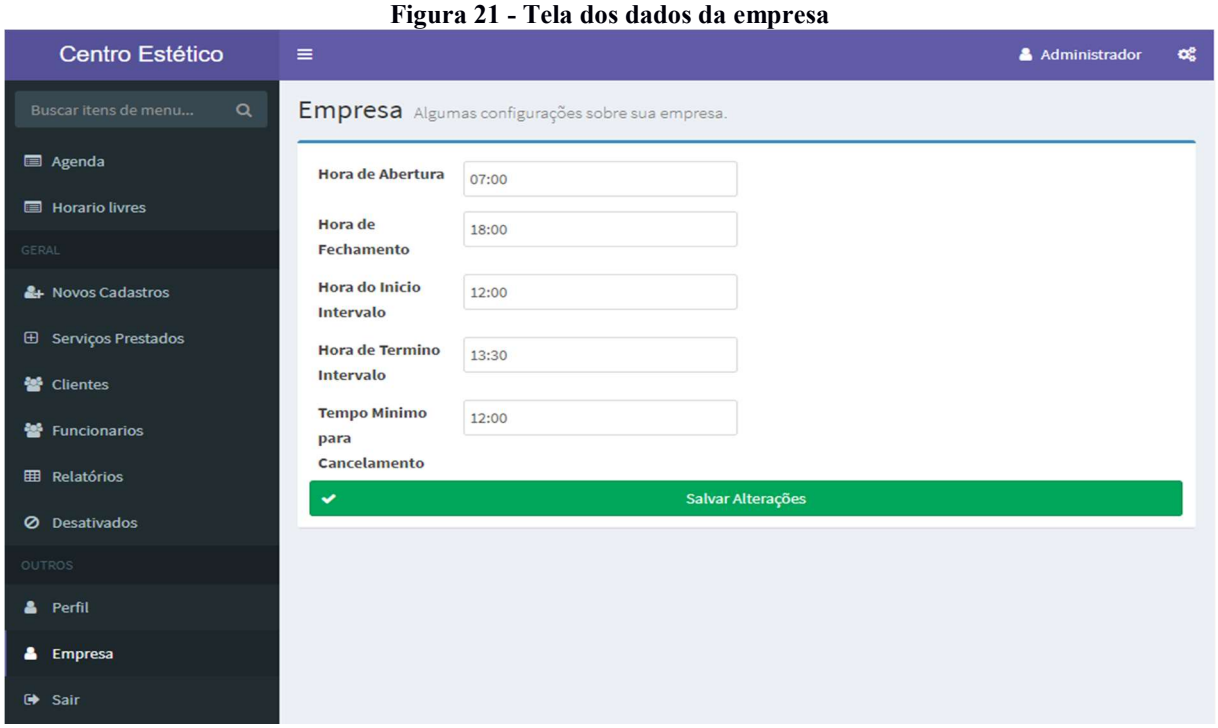
Figura 20 - Tela perfil



Fonte: autoria própria

No menu “empresa” o administrador pode alterar algumas informações sobre sua empresa, como a hora de abertura e fechamento, do início e término do intervalo e o tempo mínimo para que o cliente possa cancelar um agendamento, conforme a tela apresentada na Figura 21.

Figura 21 - Tela dos dados da empresa



Fonte: Autoria própria.

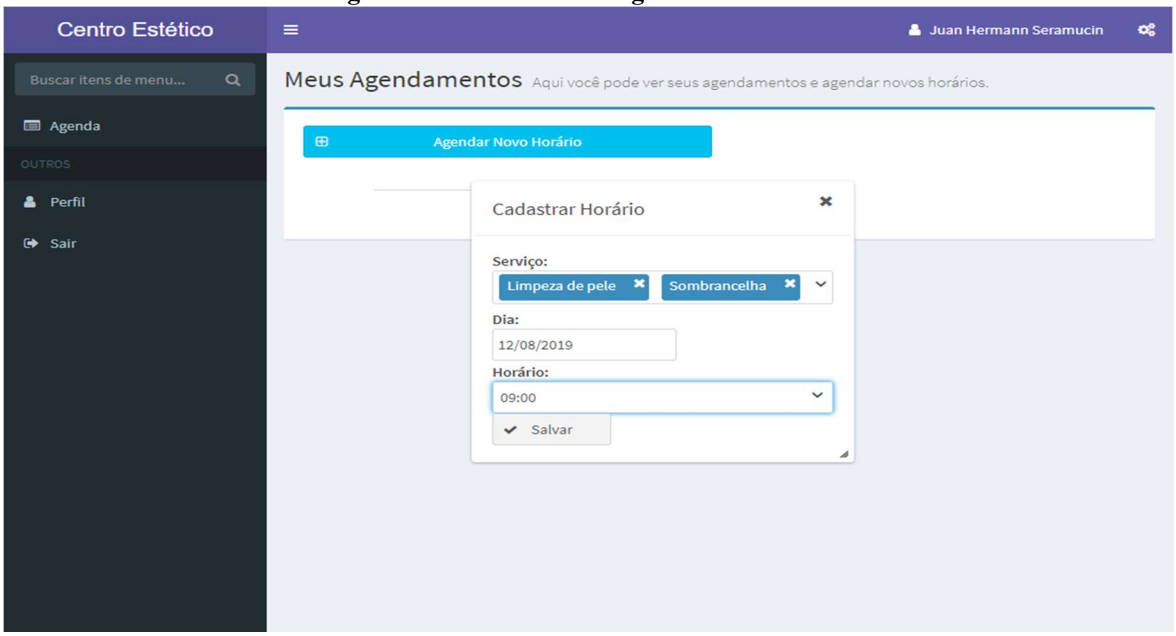
Caso o sistema seja acessado por um cliente, ele terá acesso à sua agenda e aos dados do seu perfil. A Figura 22 exibe a tela principal de clientes que não possuem agendamento.

Figura 22 - Tela principal clientes

Fonte: Autoria própria.

Ao clicar no botão “agendar novo horário” o cliente pode inserir os dados e cadastrar seu agendamento, conforme ilustra a Figura 23.

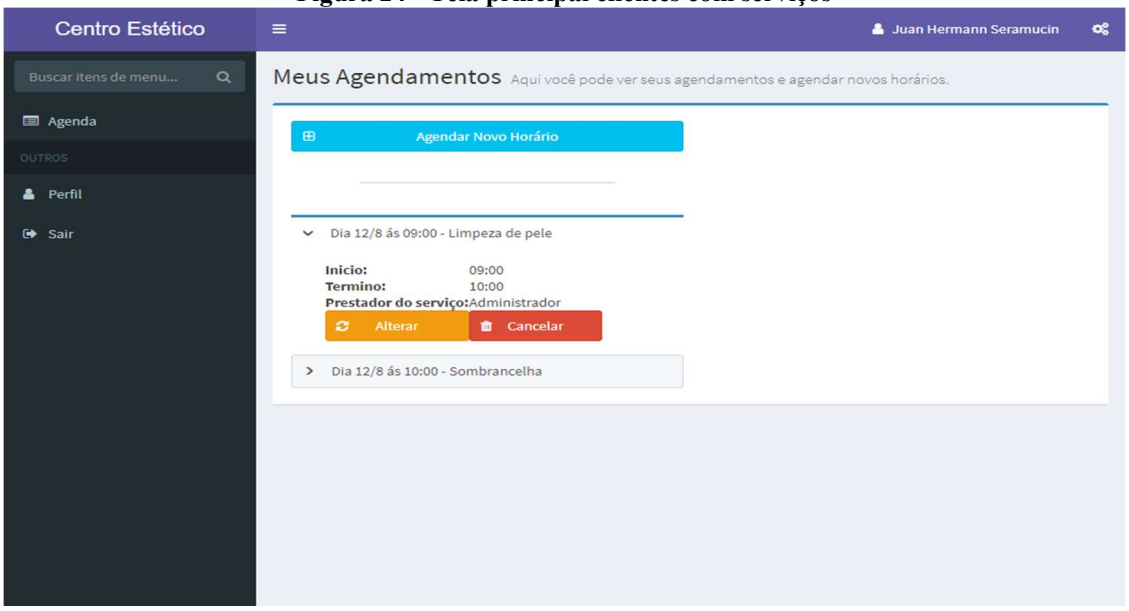
Figura 23 - Tela cadastro agendamento de clientes



Fonte: Autoria própria.

Os serviços cadastrados serão listados conforme mostra a Figura 24. O usuário pode alterar e cancelar um agendamento, caso esse esteja dentro do limite de tempo para o cancelamento estipulado pelo administrador. Nessa tela foi utilizado o componente *accordionPanel* do PrimeFaces para exibir uma listagem dos serviços, sendo que o usuário pode clicar em cima para visualizar mais detalhes sobre seu agendamento.

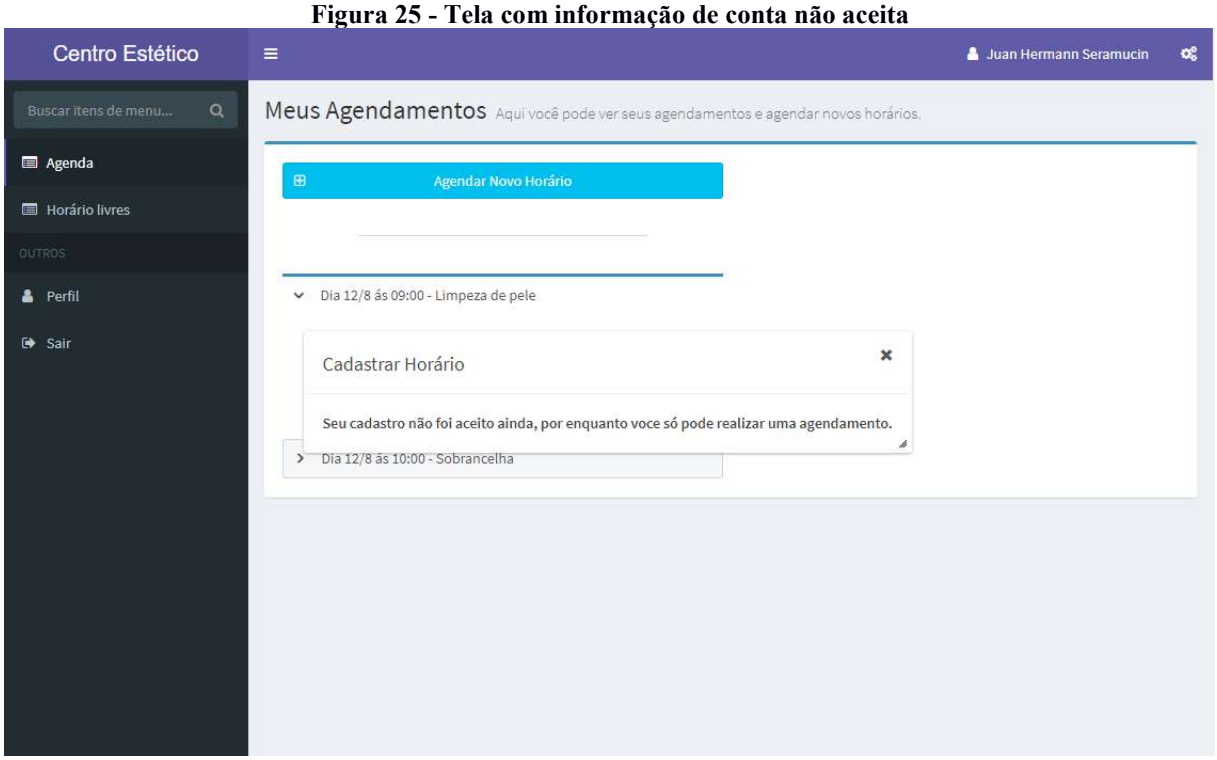
Figura 24 - Tela principal clientes com serviços



Fonte: Autoria própria.

Caso o serviço esteja dentro do horário limite para o cancelamento, não ficará disponível os botões para fazer o cancelamento ou a alteração do serviço, e caso o cadastro do cliente não tenha sido aceito até o momento e ele já tem um serviço cadastrado, quando ele tentar cadastrar um novo serviço, vai abrir o *modal* com uma mensagem, sem a possibilidade de realizar um novo cadastro de serviço, conforme mostra a Figura 25.

Figura 25 - Tela com informação de conta não aceita



Fonte: Autoria própria.

4.4 IMPLEMENTAÇÃO DO SISTEMA

A Listagem 1 exibe o código para o cadastro de um novo cliente. Foi utilizado uma *tag* <*h:form*> com campos de texto <*p:inputText*> do Primefaces para a inserção de dados do usuário e foi utilizado o componente <*p:password*> que faz a confirmação da senha automaticamente, e um <*p:commandButton*> que faz a requisição ao servidor para salvar o cadastro no banco utilizando o *framework* Spring. Também foi utilizada a *tag* <*p:messages*> para exibir as mensagens de erro e *validateRegex* para validar o campo de *e-mail*.

Listagem 1 – Código do formulário de cadastro de cliente

```
<h:form>
        <p class="login-box-msg">Informe seus dados para o Cadastro</p>
        <p:messages closable="true" />
        <div class="form-group has-feedback">
                <p:outputLabel value="Nome:" for="nome" id="lblNome" />
                <p:inputText value="#{usuarioForm.objeto.nome}" id="nome" type="text"
                        styleClass="form-control" placeholder="Informe seu nome completo"
                        required="true" requiredMessage="nome é obrigatório.">
                        <f:ajax event="change" render="@this  msgNome lblNome" />
                </p:inputText>
                <p:message id="msgNome" for="nome" display="text" />
        </div>
        <div class="form-group has-feedback">
                <p:outputLabel value="Email:" for="email" id="lblEmail" />
                <p:inputText value="#{usuarioForm.objeto.email}" id="email"
                        type="email" styleClass="form-control"
                        placeholder="Informe seu E-mail" required="true"
                        requiredMessage="Email é obrigatório."
                        validatorMessage="Informe um email valido.">
```

```
                        <f:validateRegex
                                pattern="^[_A-Za-z0-9-\+]+(\.[_A-Za-z0-9-]+)*@[A-Za-z0-9-]+(\.[A-Za-
z0-9]+)*(\.[A-Za-z]{2,})$" />
                        <f:ajax event="change" render="@this  msgEmail lblEmail" />
                </p:inputText>
                <p:message id="msgEmail" for="email" display="text" />
        </div>
        <div class="form-group has-feedback">
                <p:outputLabel value="telefone:" for="telefone" id="lblTelefone" />
                <p:inputMask id="telefone" mask="(99) 99999-9999"
                        styleClass="form-control" placeholder="Informe seu Telefone"
                        required="true" requiredMessage="Telefone é obrigatório."
                        value="#{usuarioForm.objeto.telefone}">
                        <f:ajax event="change" render="telefone  msgTelefone lblTelefone" />
                </p:inputMask>
                <p:message id="msgTelefone" for="telefone" display="text" />
        </div>
        <div class="form-group has-feedback">
                <p:outputLabel value="Senha:" for="senha" id="lblSenha1" />
                <p:password id="senha" value="#{usuarioForm.objeto.senha}"
                        styleClass="form-control" placeholder="Informe sua Senha"
                        required="false" match="pw2" redisplay="true"
                        validatorMessage="Senhas não conferem">
                </p:password>
                <i class="fa fa-lock form-control-feedback" style="font-size: 18px"></i>
                <p:message id="msgSenha1" for="senha" display="text" />
        </div>
        <div class="form-group has-feedback">
                <p:outputLabel value="Confirmação de Senha:" for="pw2" id="lblSenha2" />
                <p:password id="pw2" value="#{usuarioForm.objeto.senha}"
                        styleClass="form-control" placeholder="Confirme sua Senha"
                        required="true" redisplay="true"
                        validatorMessage="Senhas não conferem"
                        requiredMessage="Confirmação de senha é obrigatória.">
                </p:password>
                <i class="fa fa-lock form-control-feedback" style="font-size: 18px"></i>
                <p:message id="msgSenha2" for="pw2" display="text" />
        </div>
        <div class="row">
                <div class="col-xs-12">
                        <p:commandButton styleClass="btn btn-success btn-block"
                                action="#{usuarioForm.novoCadastro}" value="Criar Conta"
                                update="@form" />
                </div>
                <p:spacer height="2" />
        </div>
</h:form>
```

Fonte: Autoria própria.

A Listagem 2 exibe o método para cadastrar um novo cliente. Primeiramente ele verifica se o e-mail já foi cadastrado no sistema, e se não tiver cadastrado, continua com as informações para a pré inserção, como a criptografia da senha e a atribuição da permissão de usuário novo (ROLE_CADASTRADO). E por último foi utilizado o *Faces.redirect* do JSF para enviar o usuário a tela de autenticação.

Listagem 2 - Método para cadastrar cliente

```
public void novoCadastro() throws IOException {
   if (usuarioRepository.findByEmail(getObjeto().getEmail()) == null) {
     getObjeto().setAtivo(true);
     usuarioService.criptografarSenha(getObjeto());
     getRepository().save(getObjeto());
     getObjeto().addPermissao(permissaoRepository.findByNome("ROLE_CADASTRADO"));
     getRepository().save(getObjeto());
```

```
      addDetailMessage("Cadastro criado com sucesso!");
      Faces.getExternalContext().getFlash().setKeepMessages(true);
      Faces.redirect("index.jsf");
    } else {
      addDetailMessage("Email já cadastrado");
    }
}
```

Fonte: Autoria própria.

A geração de tabelas é um dos recursos mais utilizados para exibir uma listagem de dados. O Primefaces dispõe do componente *Datatable* que permite isso de uma forma mais rápida e organizada.

Os atributos que determinam a origem dos dados da tabela são *value* e *var.* Na Listagem 3, o atributo *value* aponta para a propriedade da classe *Bean* que fornece as informações (lista) dos serviços. O atributo *var* recebe como valor o nome da variável (serviço) que contém o valor da linha atual da tabela. Ainda foi configurado o atributo *emptyMessage* para exibir uma mensagem quando a lista de serviços estiver vazia, o atributo *rowKey* recebe a chave no *model* e o atributo *selection* que recebe a variável onde será armazenado os serviços selecionados na tabela. As demais configurações da Listagem 3 são relacionadas aos botões e para deixar a tabela como *paginator*.

Listagem 3 - Código da lista de serviços

```
<p:dataTable value="#{servicoList.lista}" var="servico"
             id="servicoDT" emptyMessage="Nenhum serviço cadastrado"
             rowKey="#{servico.id}"
             selection="#{servicoList.registrosSelecionados}" widgetVar="table"
             rows="15" paginator="true" paginatorPosition="bottom"
     paginatorTemplate="{FirstPageLink}      {PreviousPageLink}      {PageLinks}
{NextPageLink} {LastPageLink} {RowsPerPageDropdown}"
             rowsPerPageTemplate="5,10,15,20,40">
    <p:ajax event="rowSelectCheckbox" update="@(.ui-button, .ui-confirm-dialog)"
/>
    <p:ajax event="rowUnselectCheckbox" update="@(.ui-button, .ui-confirm-
dialog)" />
    <p:ajax event="rowSelect" update="@(.ui-button, .ui-confirm-dialog)" />
    <p:ajax event="rowUnselect" update="@(.ui-button, .ui-confirm-dialog)" />
    <p:ajax event="toggleSelect" update="@(.ui-button, .ui-confirm-dialog)" />
    <f:facet name="header">
        Encontre um Serviço pelo seu nome :
        <p:inputText value="#{servicoList.nome}" />
        <p:spacer height="5" />
        <p:commandButton value="Buscar" action="#{servicoList.buscar}"
                         process="@form" update="@form"
                         styleClass="btn-flat        bg-black        btn-states"
partialSubmit="true" />
    </f:facet>
    <p:column selectionMode="multiple" width="5%"
```

```
                styleClass="align-center" />
    <p:column headerText="Nome">
        <p:link value="#{servico.nome}" outcome="servico-form.xhtml">
            <f:param name="id" value="#{servico.id}" />
        </p:link>
    </p:column>
    <p:column headerText="Tempo" style="text-align: center;">
        <p:outputLabel value="#{servico.tempo}" />
    </p:column>
    <p:column headerText="Valor" style="text-align: center;">
        <p:outputLabel value="#{servico.valor}">
            <f:convertNumber pattern="R$ #,##0.00" />
        </p:outputLabel>
    </p:column>
</p:dataTable>
```

Fonte: Autoria própria.

Na Listagem 3 há cinco eventos AJAX para quando uma linha da tabela for selecionada, esse *model* seja enviado para o servidor e armazenado no *List* de registros selecionados. Além disso, o modo de seleção das caixas (*checkbox*) foi configurado para aceitar mais de uma opção (*selectionMode*="*multiple*").

Também foi utilizado o *<p:link>* para que clicando no nome do serviço o administrador seja redirecionado para a página de alteração, com a *tag <f:param>* será enviado o *id* do serviço como parâmetro para que o objeto esteja corretamente selecionado.

Na Listagem 4 foi utilizado um *AbastractListBean* que tem as funções padrões para realizar as operações de inserir, excluir e alterar. Também foi utilizado o *framework* Lombok para tornar o código menos verboso, ou seja, reduzir os códigos *boilerplate* (trechos de códigos que se são sempre os mesmos, mas não podem ser omitidos, como os métodos *getters* e *setters*). Para isso, é necessário adicionar as anotações @Getter e @Setter no topo da classe e o Lombok cria os métodos *getters* e *setters* para todos os atributos da classe.

Listagem 4 - Classe genérica para listagem

```
package com.github.adminfaces.starter.bean;

import ...

@Getter
@Setter
public abstract class AbastractListBean<M, R extends JpaRepository<M, Integer>> {

        private Integer id = 0;
        private M objeto;
        private List<M> lista;
        private final Class<M> modelClass;
        @Autowired
        private R repository;
        private List<M> registrosSelecionados;
        private String nome ="";
```

```
        AbastractListBean(Class<M> modelClass) {
                this.modelClass = modelClass;
        }

        @PostConstruct
        public void listar() {
                lista = repository.findAll();
        }

       public void novo() throws InstantiationException, IllegalAccessException {
                objeto = modelClass.newInstance();
        }

       public void remover() throws InstantiationException, IllegalAccessException {
                if (objeto == null) {
                        addDetailMessage("Nenhum registro para Excluir");

                } else {
                        repository.delete(objeto);
                        objeto = modelClass.newInstance();
                        addDetailMessage("Excluido com sucesso");
                }
        }

        public void deletarSelecionados() {
                int num =0;
                for (int i = 0; i < registrosSelecionados.size(); i++) {
                        repository.delete(registrosSelecionados.get(i));
                        num++;
                }
                registrosSelecionados.clear();
                addDetailMessage(num + " Registros deletado com sucesso!");
                listar();
        }
}
```

Fonte: autoria própria

No calendário foi desenvolvido eventos que auxiliam nas operações de um agendamento. No *schedule* o atributo *value* recebe um *ScheduleModel* como parâmetro, que conterá os agendamentos, além de algumas *tags* para a configuração visual do calendário, conforme exemplificado na Listagem 5.

Listagem 5 - Código para schedule

```
<p:schedule id="schedule" value="#{indexBean.eventModel}"
            slotMinutes="#{indexBean.tempoMinutosSchedule}"
            minTime="#{indexBean.inicioSchedule}"
            maxTime="#{indexBean.finalSchedule}" widgetVar="myschedule"
            timeFormat="HH:mm" view="agendaWeek">
    <p:ajax event="dateSelect" listener="#{indexBean.onDateSelect}"
            update="eventDetails" oncomplete="PF('inserir').show();" />
    <p:ajax event="eventSelect" listener="#{indexBean.onEventSelect}"
            update="eventDetails servico" oncomplete="PF('inserir').show();" />
    <p:ajax event="eventMove" listener="#{indexBean.onEventMove}"
            update="messages" />
    <p:ajax event="eventResize" listener="#{indexBean.onEventResize}"
            update="messages" />
```

```
</p:schedule>
```

Fonte: autoria própria

O Método addEvent() da Listagem 6, foi utilizado para atribuir os agendamentos na variável *eventModel* que será utilizado pelo Schedule para a listagem.

Listagem 6 - Método para event

```
public void addEvent() {
     if (event.getId() == null)
          eventModel.addEvent(event);
     else
          eventModel.updateEvent(event);

     event = new DefaultScheduleEvent();
}
```

Fonte: autoria própria

O método *atualizarSechedule*() da Listagem 7 é responsável por atualizar os agendamentos cadastrados. Esse método pega os dados do banco os transforma em um *DefaultScheduleEvent* que serão atribuídos a uma variável do tipo *ScheduleModel* que está sendo invocada ao *schedule*.

No calendário é atribuída uma cor diferente para um agendamento (azul) e um bloqueio (vermelho). Isso é realizado por meio do método *setStyleClass*() com o valor *btn-danger* que aplica a cor vermelha para um bloqueio. Em um bloqueio não há prestador de serviço vinculado.

O texto das variáveis que serão invocadas no *schedule* também varia de acordo com a quantidade de profissionais cadastrados, caso a quantidade seja maior que um, é adicionado no texto o nome do profissional.

Listagem 7 - Método atualizar schedule

```
public void atualizarSchedule() {
   boolean maisFuncionario = mostrarFuncionario();
   if (maisFuncionario) {
      if (funcionarioDaAgenda.getId() == null) {
         horarioAgendados = horarioAgendadoRepository.findAll();
      } else {
         horarioAgendados =
horarioAgendadoRepository.findByFuncionario(funcionarioDaAgenda.getId());
      }
   } else {
      horarioAgendados =
horarioAgendadoRepository.findByFuncionario(usuarioLogadoBean.getUsuario().getId());
```

```
   }
   eventModel = new DefaultScheduleModel();
   Date dataInicio = new Date(), dataFinal = new Date();
   for (HorarioAgendado horario : horarioAgendados) {
      LocalDateTime localDataInicio = LocalDateTime.of(horario.getData(),
horario.getHoraInicio());
      LocalDateTime localDataFinal = LocalDateTime.of(horario.getData(),
horario.getHoraTermino());
      dataInicio = Date.from(localDataInicio.toInstant(ZoneOffset.UTC));
      dataFinal = Date.from(localDataFinal.toInstant(ZoneOffset.UTC));
      String titulo = "";
      if (horario.getCliente() == null) {
         titulo = "Bloqueio";
         if (maisFuncionario) {
            titulo = titulo + " do " +
horario.getUsuarioServico().getUsuario().getNome();
         }
         DefaultScheduleEvent df = new DefaultScheduleEvent(titulo, dataInicio,
dataFinal, horario);
         df.setStyleClass("btn-danger");
         eventModel.addEvent(df);
      } else {
         titulo = horario.getCliente().getNome() + " - " +
horario.getUsuarioServico().getServico().getNome();
         if (maisFuncionario) {
            titulo = "F: " + horario.getUsuarioServico().getUsuario().getNome() + "
- C: " + titulo;
         }
         eventModel.addEvent(new DefaultScheduleEvent(titulo, dataInicio, dataFinal,
horario));
      }
   }
}
```

Fonte: autoria própria

A Listagem 8 exibe o código da tela de cadastro de serviços que foi desenvolvida com os componentes do Primefaces. O componente *SelectOneRadio* é utilizado para alterar o formulário de acordo com a opção escolhida (serviço ou bloqueio). Ambos estão inseridos em um *container* do tipo *outputPanel* e conforme a opção selecionada o formulário é alterado.

O componente *selectOneMenu* exibe um caixa de seleção e foi utilizado para selecionar o nome do cliente, horário e profissional, caso tenha mais de um cadastrado. Para selecionar os serviços foi utilizado o componente *selectCheckboxMenu* com opções de filtros e múltiplas seleções e o componente *calendar* para selecionar o dia. Além disso, a tela possui botões para salvar ou excluir um horário selecionado. O botão para excluir somente estará visível se houver algum serviço selecionado. Quando houver uma alteração nos serviços ou na data é acionado o método que busca os horários e faz a verificação da existência de horários disponíveis nessa data com esses serviços.

Listagem 8 - Código para inserir um serviço ou bloqueio

```
<p:dialog id="dlgInserir" widgetVar="inserir"
          header="Cadastrar Horário" showEffect="clip" hideEffect="clip">
    <h:panelGrid id="eventDetails" columns="1">
        <p:outputLabel for="tipo" value="Tipo:" />
        <p:selectOneRadio id="tipo" value="#{indexBean.tipo}"
                          unselectable="true">
            <f:selectItem itemLabel="Serviço" itemValue="servico"
                          id="radioServico" />
            <f:selectItem itemLabel="Bloqueio" itemValue="bloqueio"
                          id="radioBloqueio" />
            <p:ajax event="change" process="@this" update="idGeral" />
        </p:selectOneRadio>
        <p:outputPanel id="idGeral">
            <p:outputPanel rendered="#{indexBean.mostrarForm()}">
                <h:panelGrid columns="1">
                    <p:outputLabel for="cliente" value="Cliente:" id="lblCliente" />
                    <p:selectOneMenu id="cliente"
                                     value="#{indexBean.objeto.cliente}" filter="true"
                                     filterMatchMode="contains"
converter="#{usuarioConverter}"
                                     required="true">
                        <f:selectItem itemLabel="Selecione o cliente"
                                      itemValue="#{null}" />
                        <f:selectItems value="#{indexBean.clientes}" var="usuario"
                                       itemLabel="#{usuario.nome}" itemValue="#{usuario}" />
                        <p:ajax event="change" update="cliente lblCliente" />
                    </p:selectOneMenu>
                    <p:outputLabel for="servico" value="Serviço:" id="lblServico" />
                    <p:selectCheckboxMenu id="servico"
value="#{indexBean.servicosSelecionados}" label="Serviços" multiple="true" filter="true"
filterMatchMode="contains" required="true" panelStyle="width:250px">
                        <f:selectItems value="#{indexBean.servicos}" var="servico"
                                       itemLabel="#{servico.nome}" itemValue="#{servico}" />
                        <p:ajax event="change"
                                update="funcionario horario lblServico"
                                listener="#{indexBean.buscarFuncionarios()}" />
                    </p:selectCheckboxMenu>
                    <p:outputPanel rendered="#{indexBean.mostrarFuncionario()}">
                        <p:outputLabel for="funcionario" value="Profissional:" />
                        <p:selectOneMenu id="funcionario" required="false"
                                         value="#{indexBean.funcionarioDoList}"
                                         converter="#{usuarioConverter}">
                            <f:selectItems value="#{indexBean.setFuncionarios}"
                                           var="usuario" itemLabel="#{usuario.nome}"
                                           itemValue="#{usuario}" />
                            <p:ajax event="change" update="horario"
                                    listener="#{indexBean.buscarHorarios()}" />
                        </p:selectOneMenu>
                    </p:outputPanel>
                    <p:outputLabel for="data" value="Dia:" id="lblData" />
                    <p:calendar id="data" value="#{indexBean.objeto.data}"
                                required="true" pattern="dd/MM/yyyy"
                                converter="#{localDateConverter}">
                        <p:ajax event="dateSelect"
                                listener="#{indexBean.buscarHorarios}"
                                update="horario data lblData" />
                    </p:calendar>
                    <p:outputLabel value="Horário:" for="horario" id="lblHorario" />
                    <p:selectOneMenu id="horario"
                                     converter="#{localTimeConverter}" required="true"
                                     value="#{indexBean.objeto.horaInicio}">
                        <f:selectItem itemLabel="#{indexBean.stringHorario}"
```

```
                                    itemValue="" />
                        <f:selectItems value="#{indexBean.horarios}" var="horario"
                                       itemLabel="#{horario}" itemValue="#{localTime}" />
                        <p:ajax event="change" update="horario lblHorario" />
                    </p:selectOneMenu>
                </h:panelGrid>
            </p:outputPanel>
            <p:outputPanel rendered="#{!indexBean.mostrarForm()}">
                <h:panelGrid columns="1">
                    <p:outputPanel rendered="#{indexBean.mostrarFuncionario()}">
                        <p:outputLabel for="funcionarioBloqueio" value="Profissional:" />
                        <p:selectOneMenu id="funcionarioBloqueio" required="false"
                                         value="#{indexBean.funcionarioDoBloqueio}"
                                         converter="#{usuarioConverter}">
                            <f:selectItems value="#{indexBean.setFuncionariosBloqueio}"
                                           var="usuario" itemLabel="#{usuario.nome}"
                                           itemValue="#{usuario}" />
                        </p:selectOneMenu>
                    </p:outputPanel>
                    <p:outputLabel for="inicio" value="Inicio" />
                    <p:calendar id="inicio"
                                value="#{indexBean.dataInicioBloqueio}" required="true"
                                minHour="#{indexBean.inicioHoraCalendar}"
                                maxHour="#{indexBean.finalHoraCalendar}"
                                pattern="dd/MM/yyyy HH:mm"
                                stepMinute="#{indexBean.tempoMinutosSchedule}"
                                converter="#{localDateTimeConverter}" />
                    <p:outputLabel for="final" value="Final" />
                    <p:calendar id="final" value="#{indexBean.dataFinalBloqueio}"
                                required="true" minHour="#{indexBean.inicioHoraCalendar}"
                                maxHour="#{indexBean.finalHoraCalendar}"
                                pattern="dd/MM/yyyy HH:mm"
                                stepMinute="#{indexBean.tempoMinutosSchedule}"
                                converter="#{localDateTimeConverter}" />
                </h:panelGrid>
            </p:outputPanel>
            <p:commandButton value="Salvar"
            action="#{indexBean.salvarAgendamento()}" update="eventDetails schedule
messages" icon="fa fa-check" />
            <p:commandButton value="Excluir"
                             action="#{indexBean.excluirAgendamento()}"
                             update="eventDetails schedule messages" icon="fa fa-trash"
                             styleClass="btn-danger"
                             rendered="#{indexBean.objeto.id != null}" />
        </p:outputPanel>
    </h:panelGrid>
</p:dialog>
```

Fonte: autoria própria

O método *init*() verifica se o usuário que está autenticado tem permissão para acessar a página inicial e também qual é o seu nível de permissão, caso não tiver ele é redirecionado para a página do cliente, as variáveis são inicializadas e o calendário atualizado, conforme exemplificado na Listagem 9.

Listagem 9 - Método init do index

```
@PostConstruct
public void init() throws InstantiationException, IllegalAccessException {
```

```
    verificaPermissao();
    super.init();
    if (empresaRepository.findAll().size() > 0) {
        Empresa empresa = empresaRepository.findAll().get(0);
        HORA_INICIO_EMPRESA = empresa.getHoraAbertura();
        HORA_INICIO_INTERVALO = empresa.getInicioIntervalo();
        HORA_FINAL_INTERVALO = empresa.getFinalIntervalo();
        HORA_FINAL_EMPRESA = empresa.getHoraFechamento();
        TEMPO_BUSCA_ENTRE_SERVICOS = empresa.getTempoMinServico();
    }
    inicioSchedule = HORA_INICIO_EMPRESA.toString();
    finalSchedule = HORA_FINAL_EMPRESA.toString();
    tempoMinutosSchedule = TEMPO_BUSCA_ENTRE_SERVICOS.getMinute();
    inicioHoraCalendar = HORA_INICIO_EMPRESA.getHour();
    finalHoraCalendar = HORA_FINAL_EMPRESA.getHour();
    timeZoneBrasil = TimeZone.getTimeZone("America/Sao_Paulo");
    clientes = permissaoRepository.findByNome("ROLE_CLIENTE").getUsuarios();
    horarioAgendados = new ArrayList<>();
    servicos = servicoRepository.findByAtivoOrderByNome(true);
    servicos.remove(servicoRepository.findById(1).get());
    servicosSelecionados = new ArrayList<>();
    cliente = new Usuario();
    funcionario = new Usuario();
    funcionarioDoList = new Usuario();
    funcionarioDoBloqueio = new Usuario();
    funcionarioDaAgenda = new Usuario();
    funcionarios = new ArrayList<>();
    setFuncionarios = new ArrayList<>();
    setFuncionariosBloqueio                                                    =
Usuario.filtraPorRole(usuarioRepository.findByAtivoOrderByNome(true),
            "ROLE_FUNCIONARIO");
    stringHorario = "Selecione um Horário";
    carregarFuncionariosDaAgenda();
    atualizarSchedule();
}
```

Fonte: Autoria própria.

O método *onDateSelect(),* exibido na Listagem 10, é utilizado quando o usuário clicar em uma data no calendário e será aberta a janela *modal* para cadastro de um agendamento ou bloqueio. Por padrão ele abre a janela *modal* com a data selecionada no calendário e, também, com a data e hora de início do bloqueio com o dia selecionado e primeiro horário do dia e, como data e hora final de bloqueio, o primeiro dia selecionado com o último horário do dia selecionado.

Listagem 10 - Método onDataSelect

```
public void onDateSelect(SelectEvent selectEvent) {
   stringHorario = "Selecione um horario";
   setObjeto(new HorarioAgendado());
   servicosSelecionados = new ArrayList<>();
   funcionarioDaAgenda = new Usuario();
   Date dataSelecionada = (Date) selectEvent.getObject();
   event = new DefaultScheduleEvent("", dataSelecionada, dataSelecionada);

getObjeto().setData(dataSelecionada.toInstant().atZone(ZoneId.systemDefault()).t
oLocalDate());
    dataInicioBloqueio         =         LocalDateTime.of(getObjeto().getData(),
HORA_INICIO_EMPRESA);
   dataFinalBloqueio          =          LocalDateTime.of(getObjeto().getData(),
HORA_FINAL_EMPRESA);
}
```

Fonte: Autoria própria.

O método *onEventSelect*(), apresentado na Listagem 11, é utilizado quando um horário agendado é selecionado. São recuperados os dados do objeto pelo *selectEvent* por meio do método *getData*(). Além disso, é verificado se é bloqueio ou serviço para exibir na janela modal.

Listagem 11 - Método onEventSelect

```
public void onEventSelect(SelectEvent selectEvent) {
   stringHorario = "Selecione um horario";
   servicosSelecionados = new ArrayList<>();
   event = (ScheduleEvent) selectEvent.getObject();
   setObjeto((HorarioAgendado) event.getData());
   if (getObjeto().getCliente() == null) {
      tipo = "bloqueio";
      funcionarioDoBloqueio = getObjeto().getUsuarioServico().getUsuario();
      dataInicioBloqueio         =         LocalDateTime.of(getObjeto().getData(),
getObjeto().getHoraInicio());
      dataFinalBloqueio          =         LocalDateTime.of(getObjeto().getData(),
getObjeto().getHoraTermino());
   } else {
      tipo = "servico";
      funcionarioDoList = getObjeto().getUsuarioServico().getUsuario();
      servicosSelecionados.add(getObjeto().getUsuarioServico().getServico());
      buscarHorarios();
      horarios.add(getObjeto().getHoraInicio());
   }
}
```

Fonte: Autoria própria.

A Listagem 12 exibe o método *onEventMove*() que é utilizado quando o horário agendado é arrastado (por meio do *drag and drop*) para outra data ou dia do calendário. Após pegar objeto no *event*, é realizada uma soma da data, hora de início e término, com as alterações que são passadas pelo *event*. Se um agendamento foi para uma data anterior, o *event* retornará

um valor negativo, assim quando for realizada a soma, se transformará em uma subtração automaticamente.

Listagem 12 - Método onEventMove

```
public void onEventMove(ScheduleEntryMoveEvent event) {
   setObjeto((HorarioAgendado) event.getScheduleEvent().getData());
   getObjeto().setData(getObjeto().getData().plusDays(event.getDayDelta()));

getObjeto().setHoraInicio(getObjeto().getHoraInicio().plusMinutes(event.getMinut
eDelta()));

getObjeto().setHoraTermino(getObjeto().getHoraTermino().plusMinutes(event.getMin
uteDelta()));
   getRepository().save(getObjeto());
   setObjeto(new HorarioAgendado());
}
```

Fonte: Autoria própria.

A Listagem 13 exibe o método *onEventResize*(), utilizado quando é alterado o horário final de um agendamento, o código altera somente a hora do término.

Listagem 13 - Método onEventResize

```
public void onEventResize(ScheduleEntryResizeEvent event) {
   setObjeto((HorarioAgendado) event.getScheduleEvent().getData());

getObjeto().setHoraTermino(getObjeto().getHoraTermino().plusMinutes(event.getMin
uteDelta()));
   getRepository().save(getObjeto());
   setObjeto(new HorarioAgendado());
}
```

Fonte: Autoria própria.

O código da Listagem 14 exibe o método buscarHorarios() que realiza a soma do tempo total dos serviços e invoca o método horariosLivres() apresentado na Listagem 15.

Listagem 14 - Método buscar horarios

```
public void buscarHorarios() {
   tempoTotalServicos = LocalTime.of(0, 0, 0);
   for (Servico servico : servicosSelecionados) {
      tempoTotalServicos = somarLocalTime(tempoTotalServicos, servico.getTempo());
   }
   System.out.println(tempoTotalServicos);
   horariosLivres(tempoTotalServicos);
}
```

Fonte: Autoria própria.

O método que busca os horários livres, apresentado na Listagem 15, atribui para a variável “horarios” todos os horários que sejam possíveis de serem escolhidos, caso não houvesse nenhum horário agendado. Após isso ele verifica se realmente não tem nenhum horário agendado na data escolhida, se não tiver o método não traz resultados, caso contrário, ele verifica se existe tempo livre para cadastrar um novo horário entre o início até o horário cadastrado. Isso é realizado tanto no horário de manhã quanto no da tarde e, por último, ele utiliza o método retirarHorariosOcupados (Listagem 16) para retirar os horários indisponíveis para um agendamento, como por exemplo, se o horário de uma empresa é dividido de 15 em 15 minutos e há um horário agendado às 8 horas e o usuário deseja agendar um horário em que o serviço demora uma hora par ser realizado. Assim, só será possível agendar às 7 horas ou posteriormente ao serviço cadastrado, então ele pagará os horários das 7:15, 7:30, 7:45, 8 e até o serviço cadastrado ser concluído.

Listagem 15 - Métodos horários livres

```
private void horariosLivres(LocalTime TempoTotalServicos) {
   horarios = new ArrayList<>();
   LocalTime horaAuxiliar = HORA_INICIO_EMPRESA;
   while          (verificaEspacoTempo(horaAuxiliar,          TempoTotalServicos,
HORA_INICIO_INTERVALO) == true) {
      horarios.add(horaAuxiliar);
      horaAuxiliar = somarLocalTime(horaAuxiliar, TEMPO_BUSCA_ENTRE_SERVICOS);
   }
   horaAuxiliar = HORA_FINAL_INTERVALO;
   while          (verificaEspacoTempo(horaAuxiliar,          TempoTotalServicos,
HORA_FINAL_EMPRESA) == true) {
      horarios.add(horaAuxiliar);
      horaAuxiliar = somarLocalTime(horaAuxiliar, TEMPO_BUSCA_ENTRE_SERVICOS);
   }
   if (mostrarFuncionario()) {
      if (funcionarioDoList.getId() == null) {
         System.out.println("nem preferencia");
         List<LocalTime> auxHorarios = new ArrayList<>();
         auxHorarios.addAll(horarios);
         Set<LocalTime> horariosFuncionarios = new HashSet<>();
         for (Usuario funcionario : setFuncionarios) {
            if (funcionario.getId() != null) {
               horarios.clear();
               horarios.addAll(auxHorarios);
               horarioAgendados                                                    =
horarioAgendadoRepository.findByFuncionarioAndData(funcionario.getId(),
                     getObjeto().getData());
               retirarHorariosOcupados(TempoTotalServicos);
               horariosFuncionarios.addAll(horarios);
            }
         }
         horarios.clear();
         horarios.addAll(horariosFuncionarios);
         Collections.sort(horarios);
      } else {
```

```
         horarioAgendados                                                    =
horarioAgendadoRepository.findByFuncionarioAndData(funcionarioDoList.getId(),
               getObjeto().getData());
         retirarHorariosOcupados(TempoTotalServicos);
      }
   } else {
      horarioAgendados                                                       =
horarioAgendadoRepository.findByDataOrderByHoraInicio(getObjeto().getData());
      retirarHorariosOcupados(TempoTotalServicos);
   }
   if (horarios.isEmpty()) {
      stringHorario = "Nenhum Horário disponivel nesta Data";
   } else {
      stringHorario = "Selecione um horário";
   }
}

private boolean verificaEspacoTempo(LocalTime primeiroHorarioLivre, LocalTime
tempoTotalServico, LocalTime proximoServico) {

   LocalTime total = somarLocalTime(primeiroHorarioLivre, tempoTotalServico);

   if (total.isBefore(proximoServico) || total.equals(proximoServico)) {
      return true;
   }
   return false;
}
```

Fonte: Autoria própria.

A Listagem 16 apresenta o método responsável por retirar os tempos que estão ocupados. O método faz o cálculo em todos os horários cadastrados, primeiramente ele tira os horários indisponíveis para cadastro antes de o serviço começar e, por último, tira os horários que estão entre o início e término de um serviço cadastrado.

Listagem 16 - Método para retirar horários ocupados

```
private void retirarHorariosOcupados(LocalTime TempoTotalServicos) {

   LocalTime horaAuxiliar = HORA_INICIO_EMPRESA;
   if (!horarioAgendados.isEmpty()) {
      for (HorarioAgendado horarioAgendado : horarioAgendados) {
         if (verificaEspacoTempo(HORA_INICIO_EMPRESA, TempoTotalServicos,
               horarioAgendado.getHoraInicio()) == false
               &&
horarioAgendado.getHoraInicio().isBefore(HORA_INICIO_INTERVALO)) {
            horaAuxiliar = HORA_INICIO_EMPRESA;
            while (horaAuxiliar.isBefore(horarioAgendado.getHoraTermino())) {
               horarios.remove(horaAuxiliar);
               horaAuxiliar = somarLocalTime(horaAuxiliar,
TEMPO_BUSCA_ENTRE_SERVICOS);
            }
         }
```

```
         if (verificaEspacoTempo(HORA_FINAL_INTERVALO, TempoTotalServicos,
               horarioAgendado.getHoraInicio()) == false
               && horarioAgendado.getHoraInicio().isAfter(HORA_FINAL_INTERVALO))
{
            horaAuxiliar = HORA_FINAL_INTERVALO;
            while (horaAuxiliar.isBefore(horarioAgendado.getHoraTermino())) {
               horarios.remove(horaAuxiliar);
               horaAuxiliar = somarLocalTime(horaAuxiliar,
TEMPO_BUSCA_ENTRE_SERVICOS);
            }
         } else {
            horaAuxiliar = subtrairLocalTime(horarioAgendado.getHoraInicio(),
TempoTotalServicos);
            horaAuxiliar = somarLocalTime(horaAuxiliar,
TEMPO_BUSCA_ENTRE_SERVICOS);
            while (horaAuxiliar.isBefore(horarioAgendado.getHoraTermino())) {
               horarios.remove(horaAuxiliar);
               horaAuxiliar = somarLocalTime(horaAuxiliar,
TEMPO_BUSCA_ENTRE_SERVICOS);
            }
         }
      }
   }
}
```

Fonte: Autoria própria.

A Listagem 17 representa o código do gerador de relatório em *Portable Document Format* (PDF). Primeiramente ao acionar o botão de gerar relatório, ele abrirá uma nova aba com os dados necessários para o relatório no URL do navegador.

Listagem 17 - Método para abrir relatório

```
public void abrirRelatorio() throws IOException {
   LocalDate d = LocalDate.now();
   if (dataInicio == null) {
      dataInicio = LocalDate.of(d.getYear(), d.getMonth(), 1);
   }
   if (dataFinal == null) {
      dataFinal = d;
   }
   if (relatorio.equals(relatorioA)) {
      FacesContext.getCurrentInstance().getExternalContext()
            .redirect("relatorio/servicos?data1=" + dataInicio + "&data2=" +
dataFinal);
   } else if (relatorio.equals(relatorioB)) {
      FacesContext.getCurrentInstance().getExternalContext()
            .redirect("relatorio/horarios?data1=" + dataInicio + "&data2=" +
dataFinal);
   } else if (relatorio.equals(relatorioC)) {
      FacesContext.getCurrentInstance().getExternalContext()
            .redirect("relatorio/entradas?data1=" + dataInicio + "&data2=" +
dataFinal);
   } else if (relatorio.equals(relatorioD)) {
      FacesContext.getCurrentInstance().getExternalContext()
            .redirect("relatorio/funcionarios?data1=" + dataInicio + "&data2=" +
```

```
dataFinal);
   }
}
```

Fonte: Autoria própria.

Após nova aba com os dados do relatório ser aberta na URL do navegador, automaticamente o método referente ao relatório escolhido pelo usuário será acionado na nossa classe do RelatorioController, assim o método gerará o relatório e vai exibir o mesmo, como mostra a Listagem 18.

Listagem 18 - Classe controller do relatório

```
package com.github.adminfaces.starter.controller;

import ...

@Controller
@RequestMapping("/relatorio")
public class RelatorioController {

    @Autowired
    private ServicosReportService servicosReportService;

    @Autowired
    private GerarRelatorio gerarRelatorio;

    @GetMapping("/servicos")
    public void exportServicos(@RequestParam("data1")
@DateTimeFormat(pattern="yyyy-MM-dd") Date data1,@RequestParam("data2")
@DateTimeFormat(pattern="yyyy-MM-dd") Date data2,HttpServletResponse response)
throws IOException, JRException, SQLException {
       SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy");
        JasperPrint jasperPrint =
servicosReportService.generateRelatorioData("Número de Serviços
Prestados","Entre as datas "+sdf.format(data1) +" e "+ sdf.format(data2),
"classpath:/reports/ServicosProcurados.jrxml",data1,data2,"");

            gerarRelatorio.imprimir(response, jasperPrint);


    }

    @GetMapping("/horarios")
    public void exportHorarios(@RequestParam("data1")
@DateTimeFormat(pattern="yyyy-MM-dd") Date data1,@RequestParam("data2")
@DateTimeFormat(pattern="yyyy-MM-dd") Date data2,HttpServletResponse response)
throws IOException, JRException, SQLException {
       SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy");
        JasperPrint jasperPrint =
servicosReportService.generateRelatorioData("Horários Preferidos","Entre as
datas "+sdf.format(data1) +" e "+ sdf.format(data2),
"classpath:/reports/HorariosProcurados.jrxml",data1,data2,"");
```

```
        gerarRelatorio.imprimir(response, jasperPrint);


    }

    @GetMapping("/entradas")
    public void exportEntradas(@RequestParam("data1")
@DateTimeFormat(pattern="yyyy-MM-dd") Date data1,@RequestParam("data2")
@DateTimeFormat(pattern="yyyy-MM-dd") Date data2,HttpServletResponse response)
throws IOException, JRException, SQLException {
       SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy");
        JasperPrint jasperPrint =
servicosReportService.generateRelatorioData("Entrada Mensal","Entre as datas
"+sdf.format(data1) +" e "+ sdf.format(data2),
"classpath:/reports/EntradaMensal.jrxml",data1,data2,"");

        gerarRelatorio.imprimir(response, jasperPrint);


    }

    @GetMapping("/funcionarios")
    public void exportfuncionarios(@RequestParam("data1")
@DateTimeFormat(pattern="yyyy-MM-dd") Date data1,@RequestParam("data2")
@DateTimeFormat(pattern="yyyy-MM-dd") Date data2,HttpServletResponse response)
throws IOException, JRException, SQLException {
       SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy");
        JasperPrint jasperPrint =
servicosReportService.generateRelatorioData("Serviços Realizados","Entre as
datas "+sdf.format(data1) +" e "+ sdf.format(data2),
"classpath:/reports/FuncionariosServicos.jrxml",data1,data2,"classpath:/reports/
ServicosFuncionarios.jasper");

        gerarRelatorio.imprimir(response, jasperPrint);

    }
}
```

Fonte: Autoria própria.

Na parte de permissões de usuários, toda vez que uma alteração for realizada em um cadastro, é chamado o método salvarPermissoes() da Listagem 19. Primeiramente é verificado se houve alguma alteração nas permissões. A partir disso, se houve diferença é retirado todas as permissões e adicionado as permissões que ele recebeu, além disso se o usuário recebeu a permissão de profissional, é atribuído a ele todos os serviços possíveis, caso ele tenha perdido a permissão ele perde todos os vínculos com os serviços para prestar.

Listagem 19 - Método salvar permissões.

```
private void salvarPermissoes() {
   List<Permissao> lista = stringToPermissao(permissaoSelecionados);
   boolean funcionario = Usuario.hasRole("ROLE_FUNCIONARIO",
getObjeto());
   if (!getObjeto().getPermissoes().equals(lista)) {
```

```
      getObjeto().getPermissoes().clear();
      getObjeto().setPermissoes(lista);
      getRepository().save(getObjeto());
      if (!funcionario) {
         if (!permissaoSelecionados.contains("Funcionario")) {
            for (UsuarioServico us :
usuarioServicoRepository.findByUsuarioAndAtivo(getObjeto(), true)) {
               us.setAtivo(false);
               usuarioServicoRepository.save(us);
            }
         } else {
            for (Servico servico : getLista()) {
               UsuarioServico us =
usuarioServicoRepository.findByServicoAndUsuario(servico, getObjeto());
               us.setAtivo(true);
               usuarioServicoRepository.save(us);

            }
         }
      }
   }
}
```

Fonte: Autoria própria.

5 CONCLUSÃO

A realização deste trabalho permitiu adquirir novos conhecimentos abordando as tecnologias JSF e Primefaces e aperfeiçoar conhecimentos adquiridos ao longo do curso de Tecnologia em Análise e Desenvolvimento de Sistemas, como a programação que envolve o *front-end* e *back-end*, que são duas abordagens da programação *web*, a linguagem de programação Java, lógica de programação e banco de dados. Esses conteúdos são abordados desde o primeiro período do curso, na disciplina de Fundamentos de Programação até o último período em disciplinas de programação *web*.

Por meio da utilização das tecnologias Primefaces e JSF foi possível adquirir conhecimentos no desenvolvimento do *front-end* de uma aplicação *web*, criando interfaces ricas, cujos componentes são disponibilizados por meio da biblioteca do Primefaces. Foi possível observar que, utilizando essas tecnologias, o código fica mais simples, com um visual mais robusto e oferece uma melhor resposta às ações do usuário por possuir uma interface rica. Além disso, o desenvolvimento desse tipo de aplicação utilizando essas tecnologias, permite executar processamento assíncrono, sem a necessidade de recarregar a página e redução da complexidade das aplicações. A documentação das ferramentas e tecnologias foi bem significativa para o estudo durante o desenvolvimento, pois é bem completa e com exemplos bem definidos.

Por fim, o desenvolvimento deste trabalho também possibilitou aprimorar o raciocínio lógico no desenvolvimento das funcionalidades que estão no servidor, para realizar as operações e transições dos dados do banco de dados.

Como trabalho futuro, pode-se desenvolver aplicativos nativos, para ter um aproveitamento ainda melhor das funcionalidades que um celular pode proporcionar. Além de continuar a atualização do projeto para proporcionar eficiência e produtividade para o usuário na utilização do sistema.

REFERÊNCIAS

ABIHPEC. **MEIs do mercado de beleza crescem 567% em 5 ano**. 2015.

CGI.BR. **Cresce a proporção de empresas brasileiras que utilizam conexões à Internet mais velozes, aponta Cetic.br**. 2016. Disponível em: https://www.cgi.br/noticia/releases/cresce-a-proporcao-de-empresas-brasileiras-que-utilizam-conexoes-a-internet-mais-velozes-aponta-cetic-br. Acesso em: 26 mar. 2018.

COELHO, Hebert. **JSF Eficaz**: as melhores práticas para o desenvolvedor web Java. São Paulo: Casa do Código, 2012.

CORDEIRO, Gilliard. **Aplicações Java para a web com JSF e JPA**. São Paulo: Casa do Código, 2014.

COULOURIS George; DOLLIMORE Jean; KINDBERG Tim, BLAIR Gordon. **Sistemas Distribuídos** - 5ed: Conceitos e Projeto. Editora: Bookman, 2013.

Euromonitor International. **Na hora de buscar crescimento fora do Brasil**, 2018. Disponível em: https://blog.euromonitor.com/2017/10/hora-buscar-crescimento-brasil.html. Acesso em: 28 mar. 2018.

FOWLER, Martin. **Patterns of Enterprise Application Architecture**, Addison-Wesley Professional, 2003.

FRATERNALI Piero; Rossi Gustavo; Sánchez-Figueroa Fernando. Rich Internet Applications. **IEEE Computer Society**, 2010.

FUENTES Vinícius Baggio. **Ruby on Rails**: Coloque sua aplicação web nos trilhos. Casa do Código: São Paulo, 2014.

GEARY, David; HORSTMANN, Cay. **Core JavaServer Faces**. Rio de Janeiro: Alta Books, 2007.

GORLA, João Paulo F.; FOSCHINI, Ivan João. Arquitetura para desenvolvimento web baseado em JSF 2.0 utilizando Padrões de Projeto. **T.I.S. Tecnologias, Infraestrutura e Software** v.2, n.3, 2013. Disponível em http://revistatis.dc.ufscar.br/index.php/revista/article/*view*/68 . Acesso em: 14 mai. 2019.

LIMA, Vinicius F., MAGALHÃES, Willian B. Desenvolvimento web com JSF e os escopos *view*, request, session, application e controle de transações com EJB. **XVII SEINPAR**. Semana de Informática de Paranavaí, 2014. Disponível em: http://web.unipar.br/~seinpar/2014/artigos/graduacao/vinicios_fernandes_de_lima.pdf. Acesso em: 14 mai. 2019.

MELIÁ Santiago; GÓMEZ Jaime; PÉREZ Sandy; DÍAZ Oscar. Facing Architectural and Technological Variability of Rich Internet Applications. **IEEE Internet Computing**, 2010.

MILETTO Evandro Manara; BERTAGNOLLI Silvia de Castro. **Desenvolvimento de Software II**: Introdução ao Desenvolvimento Web com HTML, CSS, JavaScript e PHP. Editora: Bookman, 2014.

PINA, D.S.A., OLIVEIRA, L. E. M. C. RIA - **Rich Internet Applications**: Uma Revisão dos Principais Expoentes da Área, União dos Institutos Brasileiros de Tecnologia. Recife, 2013.

PRESSMAN, Roger. **Engenharia de software**. Rio de Janeiro: McGraw-Hill, 2008.

SILVA, Mateus Antontio C.; FOSCHINI, Ivan João. Implementação do padrão Façade utilizando o framework JavaServer Faces: um estudo de caso. **T.I.S. Tecnologias, Infraestrutura e Software** v.1, n.1, 2012. Disponível em http://revistatis.dc.ufscar.br/index.php/revista/article/*view*/11/14. Acesso em: 14 mai. 2019.

SOUZA, Alberto. **Spring MVC**: Domine o principal framework *web* Java. Editora: Casa do Código, 2015.