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

MATHEUS TONIAL

SISTEMA WEB PARA CONTROLE E GERENCIAMENTO DE PAGAMENTOS DE SEGURO

TRABALHO DE CONCLUSÃO DE CURSO

PATO BRANCO
2019

MATHEUS TONIAL

SISTEMA WEB PARA CONTROLE E GERENCIAMENTO DE PAGAMENTOS DE SEGURO

Trabalho de Conclusão de Curso de graduação, apresentado à disciplina de Trabalho de Conclusão de Curso 2, do Curso 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: Profa**.** Andreia Scariot Beulke

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 PAGAMENTOS DE SEGURO

POR

MATHEUS TONIAL

Este trabalho de conclusão de curso foi apresentado no dia 27 de junho 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
Beatriz Terezinha Borsoi

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

Profª Drª Beatriz Terezinha Borsoi
Responsável pela Atividade de Trabalho de Conclusão de Curso

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

RESUMO

Os sistemas web estão cada vez mais comuns na vida das pessoas, seja pelo uso profissional ou pessoal. Esses sistemas permitem acesso às informações a qualquer momento desde que se tenha acesso à Internet, permitindo, assim, que as operações comerciais sejam realizadas de forma mais rápida e que atinjam um número maior de usuários. Visando potencializar o controle e o gerenciamento das operações de pagamentos de parcelas de seguros e dos seguros a vencer de uma corretora de seguros, este trabalho visa o desenvolvimento de um sistema *web* que permite o acesso por meio de dispositivos diversificados. Por meio do sistema, o corretor terá acesso às informações necessárias para consultar os pagamentos de parcelas e dos seguros a vencer, evitando, assim, o cancelamento ou multas pelo atraso no pagamento, pois poderá entrar em contato com o cliente para informá-lo sobre os vencimentos. Além disso, o sistema enviará mensagem pelo correio eletrônico informando o cliente sobre os vencimentos e o cliente poderá consultar seus pagamentos cadastrados. Dentre as tecnologias que foram utilizadas no trabalho estão a linguagem Java, o *framework* Spring para a instanciação de classes e o Spring Security, responsável pela segurança da aplicação.

Palavras-chave: Corretora de Seguros. Sistema Web. Java. Spring.

ABSTRACT

The web systems are increasingly common in people's lives, whether by professional or personal use. These systems are accessible at any time as long as it has internet access, therefore enabling business operations to be performed more quickly and reach a larger number of users. Aiming at enhancing the control and management of operations of payment of insurance installment and insurance to be expiring, of an insurance company, this work aims to develop a web system that allows access through diversified devices and at any time. Through the system, the broker will have access to the necessary information to consult installment payments and insurance to expire, then, avoiding cancellation or penalties for late payment, because it will be possible to contact the client to warn him about the expiration date. In addition, the system will send a message by e-mail informing the client about the expiration date and the client can check their registered payments. Among the technologies to be used in the work, are the Java language, the Spring framework used for class instantiation and Spring Security, responsible for the security of the application.

Keywords: Insurance Company. Web System. Java. Spring.

LISTA DE FIGURAS

LISTAGEM DE QUADROS

LISTA DE SIGLAS

CGI	Comitê Gestor da Internet
MVC	*Model View Controller*
PDF	*Portable Document Format*
SI	Sistemas de Informação
XML	Extensible Markup Language

SUMÁRIO

1 INTRODUÇÃO

Este capítulo apresenta as considerações iniciais, os objetivos e a justificativa da realização deste trabalho e a organização do texto com a apresentação dos capítulos que compõem o texto.

1.1 CONSIDERAÇÕES INICIAIS

A utilização de sistemas *web* é cada vez mais comum na vida das pessoas devido à facilidade de acesso aos conteúdos disponibilizados. De acordo com Berners-Lee (1994) a tecnologia *web* surgiu para formar um repositório de conhecimentos humanos por meio do armazenamento, recuperação e visualização de documentos eletrônicos.

Os sistemas *web* permitem acesso aos vários tipos de informação, seja empresarial, pessoal ou institucional e, além disso, permitem a comunicação entre diversos ambientes, proporcionando, assim, oportunidade e versatilidade nos negócios. Esses sistemas possibilitam realizar rotinas e processos comerciais, industriais e de serviços permitindo mobilidade corporativa e facilidade para as pessoas realizarem suas atividades pessoais e profissionais.

Dados do Comitê Gestor da Internet (CGI) apontam que 43% das empresas utilizam algum software que foi desenvolvido para atender à demanda e às necessidades específicas da empresa. Os dados também revelam que os três principais motivos citados pelas empresas para introdução de softwares novos ou que passaram por algum aperfeiçoamento são: melhoria de procedimentos internos (35%), ganho de produtividade (22%) e adequação às exigências legais (19%) (CGI, 2013).

Em relação ao comércio de seguros, os Sistemas de Informação (SI) podem auxiliar no gerenciamento e na gestão dos seus processos e tarefas. As tecnologias de informação e comunicação têm contribuído com as seguradoras por meio dos SI pela automatização dos processos técnicos e administrativos.

Para atingir seus objetivos, uma empresa de seguros precisa focar na sua carteira de clientes sem deixar de lado a gestão dos negócios. Nesse sentido, automatizar recursos que permitem obter informações de forma rápida e segura visando agilizar o atendimento aos clientes ou resolver problemas é importante para estabelecer um vínculo mais confiável com o cliente.

Considerando esse contexto, este trabalho visa o desenvolvimento de um sistema *web* que permita controlar e gerenciar os pagamentos das parcelas dos seguros e do próprio seguro, visando evitar o cancelamento ou multas pelo atraso no pagamento.

1.2 OBJETIVOS

A seguir são apresentados o objetivo geral e os objetivos específicos do sistema proposto neste trabalho. O objetivo geral está relacionado ao sentido mais amplo da realização deste trabalho e os objetivos específicos complementam o geral por meio de funcionalidades do sistema.

1.2.1 Objetivo Geral

Desenvolver um sistema *web* para controle e gerenciamento de pagamentos de seguro.

1.2.2 Objetivos Específicos

Para alcançar o objetivo geral, os seguintes objetivos específicos foram definidos:

- Possibilitar que o corretor possa gerenciar e controlar os vencimentos das parcelas e dos seguros a vencer de seus clientes.
- Viabilizar a emissão de relatórios das parcelas e dos seguros pagos e a vencer.
- Oportunizar a consulta para o cliente dos pagamentos de parcelas e dos seguros realizados.
- Possibilitar o envio de *e-mails* para os clientes notificando-os sobre o vencimento das parcelas e dos seguros a vencer.

1.3 JUSTIFICATIVA

Os corretores de seguros são intermediários entre as seguradoras e os clientes legalmente autorizados para vender seguros. Com relação às vendas, sua função é negociar contratos de seguros para pessoas que procuram proteção para o seu patrimônio, saúde ou

vida (SEBRAE, 2018). Dentre as funções desenvolvidas pelos corretores de seguros estão a negociação de valores, o estabelecimento de contrato (apólice) e o controle dos pagamentos das parcelas e do seguro.

Quando o cliente realiza a compra de um seguro o valor pode ser dividido em parcelas, sendo os pagamentos realizados por meio de boleto ou débito em conta. Quando ocorre o vencimento da primeira parcela e o cliente não realizou o pagamento, o seguro é automaticamente cancelado, anulando a apólice do seguro realizado e o processo de efetivação precisa ser realizado novamente, devendo ser refeitas as cotações e a proposta do seguro. Após esse procedimento as companhias têm quinze dias corridos de prazo para avaliar a proposta e gerar a nova apólice. A partir da segunda parcela, caso o cliente não efetue o pagamento, o seguro não é cancelado, contudo, ocorre a geração de multas. O valor da multa depende dos critérios estabelecidos pela companhia de seguros que gerou a apólice.

Para evitar que o cliente fique desprovido do seguro e o corretor precise refazer cotações e proposta, é necessário que o corretor entre em contato com o cliente para lembrá-lo do prazo do pagamento. Para isso, é necessário ter os dados da apólice de todos os clientes. Esse trabalho torna-se moroso e suscetível a erros se realizado de forma manual ou mesmo por planilhas eletrônicas.

Nesse sentido, este trabalho se justifica pela proposta de desenvolvimento de um sistema *web* que possa auxiliar os corretores no controle e gerenciamento das parcelas dos seguros e do próprio seguro por meio da centralização das informações em banco de dados. Ainda, permite que o cliente tenha acesso aos dados dos seus seguros por meio de autenticação no sistema.

Dentre as principais tecnologias que foram utilizadas no desenvolvimento do trabalho proposto estão a linguagem Java e o *framework* Spring cuja funcionalidade básica é a instanciação de classes, realização de dependências com base em definições em um arquivo de configuração *Extensible Markup Language* (XML) criado pelo desenvolvedor (SOUZA, 2015). O Spring *Security* permite deixar o sistema mais seguro, pois somente quem é cadastrado no sistema poderá acessar as funcionalidades restritas. Também, é importante que o leiaute do sistema se adapte em qualquer tamanho de dispositivo, para isso, foi trabalhado com os recursos de *media queries* fornecidos pelo *framework* Bootstrap.

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 e o desenvolvimento do sistema. Por fim, está a conclusão do trabalho.

2 APLICAÇÃO WEB

Este capítulo apresenta a fundamentação teórica deste trabalho, cujo conteúdo explana sobre os conceitos de arquitetura de sistemas *web*, arquitetura *Model View Controller* (MVC), requisições síncronas e assíncronas e os padrões de desenvolvimento *web* e sobre o desenvolvimento *web* tradicional.

2.1 ARQUITETURA EM CAMADAS

O termo arquitetura em camadas é utilizado para designar que um software é dividido em camadas, sendo que cada uma delas possui uma função específica, seja interface com usuário, regras de negócio ou banco de dados. De acordo com Martins (2007, p.18), cada camada aborda um aspecto técnico da arquitetura e o software pode ter de uma a n camadas.

A arquitetura mais conhecida é a de duas camadas (cliente e servidor). Sua funcionalidade depende do computador do cliente efetuar uma conexão estável com o computador servidor. Segundo Ramakrishnan e Gehrke (2008, p. 197) podem-se dividir os clientes desta arquitetura entre cliente magro e cliente gordo. O cliente magro executa somente a interface gráfica, deixando a funcionalidade de lógica e gerenciamento de dados a cargo do servidor. O cliente gordo é responsável pela interface e, também, pela lógica de negócio e o servidor fica somente com a responsabilidade do gerenciamento de dados.

A vantagem de se utilizar esta arquitetura vem principalmente do nível de confiança da rede, como o servidor é o ponto central, ele gerencia recursos compartilhados como o banco de dados centralizado, evitando problemas de redundância e contradições, possuindo um número menor de pontos de entrada aos dados, podendo adicionar e remover registros sem prejudicar o funcionamento da rede. Garante, também, maior segurança, pois há menos pontos de acesso aos dados. Porém, como todo o sistema é criado em torno do servidor, ele se torna sua maior desvantagem e necessita de *backups* e um controle cuidadoso.

A arquitetura de três ou mais camadas é similar à de duas camadas. A diferença consiste que pode haver mais de um servidor no meio da operação. O cliente em sua interface gráfica (primeira camada) faz uma requisição que é atendida pelo servidor do meio (segunda camada) que se responsabiliza por proporcionar o que foi requisitado pelo cliente e o servidor de dados (terceira camada) que fica responsável por fornecer os dados ao servidor intermediário.

Para Martins (2007) a escolha do padrão de arquitetura e a divisão em camadas mais adequadas ao software dependem de um conjunto de características necessárias ao projeto como concorrência, persistência e distribuição. A concorrência se refere à execução de várias tarefas em paralelo e cada processamento concorrente pode ser implementado com um padrão de arquitetura diferente. A persistência se refere aos dados que são armazenados em banco de dados ou arquivos e a distribuição está associada na forma em como os componentes são alocados nos computadores da rede e como eles se comunicam entre si. Essas características são integradas por meio da comunicação entre as camadas que é baseada entre o cliente e o servidor. O cliente aciona o servidor sempre que precisa fazer comunicação que deve estar sempre disponível para receber as requisições. Nesse tipo de comunicação, um ou mais clientes podem acessar o mesmo servidor, que permite o processamento distribuído, análise e apresentação dos resultados (RODRIGUEZ; FERRANTE, 1995, p. 293).

Existem modelos (ou padrões) de desenvolvimento de aplicações para trabalhar com as arquiteturas. Um deles é o MVC que é utilizado no modelo de arquitetura três camadas.

2.1 ARQUITETURA MVC

O MVC é muito utilizado no desenvolvimento de aplicações *web*, devido à sua estrutura que auxilia na divisão de camadas de uma aplicação, separando-a em *model*, *view* e *controller*. Facilitando, assim, a separação da interface gráfica, do domínio e do negócio (SILVEIRA et al., 2011, p. 280).

A camada *model* permite o acesso aos dados e às classes do sistema, é ela que manipula as informações e realiza consultas no banco de dados.

A camada *view* é responsável pela apresentação da interface do sistema. Nessa camada são implementados, apenas, os recursos gráficos, como componentes, botões e mensagens.

O *controller* é a camada intermediária que recebe a função de conectar os dados, pegando as informações lidas do usuário e enviando-as para serem tratadas pelo sistema, executando o que foi estipulado no modelo e enviando as informações para a visão. Outra importante funcionalidade dessa camada é garantir a segurança dos registros de dados, detectando qualquer informação.

Entre as principais vantagens do modelo MVC estão: a possibilidade de reescrita da interface gráfica baseada em usuário ou do controlador sem alterar o modelo; a reutilização da

interface gráfica em outras aplicações; a facilidade de manutenção e recursos; reutilização de códigos; entre outros (DOOLEY, 2011).

Após definir o tipo de arquitetura utilizada no desenvolvimento de aplicações *web*, *desktop* ou *mobile*, é importante compreender como são realizadas as requisições entre o cliente e o servidor, pois o cliente realiza uma requisição por meio de uma interface gráfica e o servidor a recebe, realiza o processamento e devolve a resposta para o cliente. Ao utilizar o modelo MVC, o *controller* é quem irá fazer o gerenciamento das requisições entre a *view* e a *model*.

2.2 REQUISIÇÕES SÍNCRONA E ASSÍNCRONA

Em aplicações *web* e distribuídas as requisições podem ser compreendidas como síncrona e assíncrona. Na comunicação síncrona, o emissor e o receptor da mensagem devem se conectar e permanecer em sincronia durante toda a sua transmissão, pois esse meio de comunicação é dependente de vários fatores, como, por exemplo, o tempo entre as mensagens e a ordem em que elas são transmitidas. Durante sua transferência o emissor não pode enviar outra mensagem, ele é obrigado a esperar a primeira chegar ao receptor, dependendo totalmente de que a transmissão permaneça intacta.

Na comunicação assíncrona não tem a necessidade de esperar uma mensagem chegar ao seu receptor para que outra seja enviada. É possível o envio de várias mensagens simultaneamente, sendo assim, as respostas também não são recebidas na mesma ordem. A transmissão assíncrona possui um *bit* especial que é inserido antes e depois de cada mensagem, podendo ser enviadas fora de ordem e quando chegarem ao receptor, esse bit é interpretado e as mensagens são colocadas em ordem (SCHUNCKE, 2013).

Comer (2016, p.142) simplifica dizendo que na comunicação assíncrona a transmissão pode acontecer a qualquer momento e o sistema pode permanecer inativo durante as transmissões e na comunicação síncrona os *bits* são transmitidos de forma constante e os dados são agrupados em pacotes.

Em uma aplicação *web* as requisições síncronas e assíncronas são realizadas por meio de um objeto chamado XMLHttpRequest. Na requisição síncrona, o *browser* ficará parado até que a requisição seja completada e, na assíncrona, após e requisição ser enviada uma função *callback* será invocada (SAMPAIO, 2007, p. 76). *Callbacks* são porções de código que

serão executados no momento adequado. Sempre que uma tarefa assíncrona entra em ação, a função *callback* é registrada e fica esperando uma mensagem para ser executada.

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 utilizadas no desenvolvimento do sistema proposto.

Quadro 1 – Lista de ferramentas e tecnologias utilizadas

Ferramenta / Tecnologia	Versão	Disponível em	Aplicação
Apache Maven	3.3.9	https://maven.apache.org	Gerenciamento de dependências e *build*.
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.*
IntelliJ IDEA	2018.2	https://www.jetbrains.com/idea	Ambiente de desenvolvimento para o sistema.
Java	8	http://www.java.com/pt_BR/download/win10.jsp	Linguagem de programação.
Lombok	0.20.2018.2	https://projectlombok.org/	Biblioteca Java focada em produtividade e redução de código redundante.
pgAdmin III	1.22.2	https://www.pgadmin.org	Ferramenta para administração do banco de dados.
PostgreSQL	9.5	https://www.postgresql.org	Banco de dados.
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.
Jaspersoft Studio	6.4.3	https://community.jaspersoft.com/download	Gerador de relatórios.

Fonte: Autoria própria

3.2 MÉTODO

O método consiste nas atividades de levantamento e modelagem de requisitos. 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. Neste trabalho são apresentadas as duas primeiras etapas.

Na fase inicial do levantamento de requisitos foi realizada entrevista com uma corretora de seguros para compreender os processos e as atividades realizadas para efetivação dos seguros. A corretora utilizada como estudo de caso para o desenvolvimento desse sistema é representante de uma franquia de seguros e possui sistemas próprios das companhias de seguros para realização de cotações, propostas e efetivação dos seguros. Esses sistemas fornecem todos os recursos necessários para registros dos clientes e dos seguros realizados. Porém, levantou-se a necessidade de um controle via sistema dos pagamentos dos seguros, seja para a efetivação de um novo ou renovação, além das parcelas pagas e a pagar. Assim, tanto o corretor quanto o cliente podem consultar os pagamentos e o corretor pode gerar relatórios dos seguros e das parcelas a vencer.

Na fase de análise e projeto do sistema tendo como base os requisitos pré-estabelecidos, 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 etapa da implementação o sistema foi codificado utilizando as tecnologias e ferramentas apresentadas no Quadro 1.

Os testes foram realizados informalmente à medida em que cada funcionalidade do sistema foi desenvolvida.

4 RESULTADOS

Este capítulo apresenta o resultado deste trabalho que é o desenvolvimento de um sistema *web* para controle e gerenciamento de pagamentos de seguros, incluindo as parcelas, a efetivação e a renovação do seguro.

4.1 ESCOPO DO SISTEMA

Uma corretora é responsável por efetivar e renovar seguros de diversos tipos que são realizados anualmente. Ao contratar um seguro, o proponente adquire um plano de adesão por meio de um contrato intermediado pelo corretor de seguro.

O sistema possui as funcionalidades de cadastro de usuários (cliente e corretor), do seguro e dos tipos de seguro. O sistema também possui o cadastro de veículos, pois são os seguros realizados em maior número e, além disso, uma pessoa pode ter mais de veículo. O corretor é a pessoa que acessa todas as funcionalidades do sistema. Ele terá acesso a todos os cadastros e poderá emitir relatórios relacionados aos seguros e ao pagamento das parcelas. O cliente poderá visualizar em uma listagem os dados de todos os seus seguros.

Ao efetivar um seguro, o corretor deverá autenticar-se no sistema e deverá cadastrar o cliente e o seguro com os dados do seguro e dos pagamentos. Na tela de cadastro de seguros, o corretor deverá informar o tipo do seguro, o nome do cliente, o valor, a quantidade de parcelas e a data da efetivação do seguro.

As funcionalidades referentes às operações de cadastro são padronizadas para todo o sistema. Para a opção de inclusão ou alteração de um registro é exibida uma janela *modal* com os campos de entrada. Ao selecionar uma opção de cadastro no menu principal é exibida uma tela com a listagem dos dados referentes à opção selecionada e, a partir dessa tela, poderá ser realizado o cadastro de um novo registro, a exclusão, a alteração e a busca por um registro específico.

Na tela de seguros será possível visualizar os dados do cliente (nome e *e-mail*) o tipo de seguro realizado, a data de efetivação, o valor e a quantidade de parcelas. Essa informação possui um botão para acessar os dados de cada parcela (o *status* e a data do pagamento). Ainda, nessa tela, o corretor poderá notificar o cliente sobre o vencimento das parcelas e dos seguros, com um botão que fica disponível conforme os prazos configurados.

O corretor poderá emitir relatórios dos seguros e das parcelas a vencer, dos seguros de um cliente, das parcelas vencidas e por tipo de seguro.

O cliente poderá visualizar os dados dos seus seguros. Nessa tela, não será possível fazer alterações, somente visualizar os dados dos pagamentos, podendo conferir quantas parcelas foram pagas, quantas faltam e a data de pagamento das próximas parcelas.

4.2 MODELAGEM DO SISTEMA

Esta seção apresenta os requisitos funcionais e não funcionais e os diagramas desenvolvido para detalhar os processos e a estrutura do software.

Nos quadros a seguir os requisitos funcionais estão representados com a sigla RF e os requisitos não funcionais com a sigla RNF, cada um com seu nome e descrição. No Quadro 2 estão os requisitos funcionais do sistema.

Quadro 2 – Requisitos funcionais

Identificação	Nome	Descrição
RF1	Manter usuários	Manutenção dos dados relacionados aos usuários do sistema (corretor e cliente).
RF2	Manter categorias de seguros	Manutenção das categorias dos seguros. Um seguro pode ser auto (para veículos), residencial, comercial, de vida, frotas, aeronáutico, entre outros.
RF3	Manter veículos	Manutenção dos dados do veículo (placa, modelo, ano e marca).
RF4	Manter marcas	Cada veículo está relacionado com uma marca.
RF5	Manter seguros	Cada seguro está relacionado a um cliente e a um tipo de seguro. Se o seguro for de veículo é necessário informar a placa do veículo (do cadastro de veículo). Também é necessário informar o número de parcelas, o valor e a data do seguro (seleciona a data atual, sendo possível alterar).
RF6	Emitir relatórios	Relatórios relacionados aos dados do seguro, das parcelas, do cliente e de tipos de seguro.
RF7	Visualizar dados do seguro	O cliente poderá visualizar dados dos pagamentos dos seus seguros, como, parcelas pagas, a vencer, vencimento do seguro, entre outras.
RF8	Personalizar notificação	O corretor poderá personalizar a notificação do vencimento dos seguros e das parcelas que é enviada por *e-mail* para o cliente, informando o título, o texto e os prazos para a notificação. Esses dados possuem preenchimento padrão, sendo possível alterar, conforme a necessidade.

Fonte: Autoria própria

No Quadro 3 estão os requisitos não funcionais do sistema e estão relacionados ao acesso ao sistema, validação de campos e de relatórios.

Quadro 3 – Requisitos não funcionais

Identificação	Nome	Descrição
RNF1	Efetuar *login*	O sistema validará *login* e senha para conceder acesso ao sistema, sendo que o *login* (e-mail) e senha serão validos a partir do cadastro do usuário no sistema. O administrador validará o cadastro de cliente.
RNF2	Campos de preenchimento obrigatórios	Os campos que são de preenchimento obrigatório serão validados por meio de uma função do sistema.
RNF3	Campos com máscaras de entrada	Os campos que possuem caracteres especiais e, que não serão armazenados no banco de dados serão validados por meio de máscaras de entrada.
RNF4	Vínculo	Dados relacionados devem estar vinculados. Por exemplo, uma categoria deve estar relacionada a um seguro. O sistema também não deve permitir a exclusão de dados vinculados (não é possível excluir uma categoria se possuir seguros relacionados).

Fonte: Autoria própria

No Quadro 4 é descrita a operação do tipo "manter" que refere-se à inclusão dos registros de usuários, categoria de seguro, veículo e seguros.

Quadro 4 – Operação "incluir" dos casos de uso de cadastro

Caso de uso:
Incluir (refere-se à operação de inclusão de todos os casos de usos identificados como "manter").

Descrição:
Inclusão dos dados cadastrais no sistema.

Evento Iniciador:
Ator solicita inclusão de um registro no sistema.

Atores:
Administrador ou corretor.

Pré-condição:
Estar autenticado no sistema.

Sequência de Eventos:
1. Ator seleciona a opção de cadastro.
2. O sistema exibe a tela com os campos a serem preenchidos.
3. Ator preenche os campos com os respectivos dados.
4. Ator confirma a ação clicando no botão "Salvar.
5. Sistema salva os dados, mostra um aviso de confirmação e volta para a página anterior.

Pós-Condição:

Registro inserido no banco de dados. **Extensões:** Informações provenientes de outros cadastros.	
Nome do fluxo alternativo (extensão)	Descrição
1. Campos obrigatórios não informados.	1.1. O cliente 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.
2. Campos informados em formato incorreto.	2.1. O cliente informa dados 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 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:** Alteração dos dados cadastrais no sistema. **Evento Iniciador:** Ator solicita alteração de um registro no sistema. **Atores:** Administrador ou corretor, de acordo com suas funções definidas no caso de uso. **Pré-condição:** Registro estar incluso no sistema. **Sequência de Eventos:** 1. Ator acessa a tela para visualização dos dados do registro. 2. O sistema apresenta o registro selecionado para alteração. 3. Usuário altera os dados do registro. 4. O sistema altera as informações no banco de dados e informa ao usuário o *status* do procedimento. **Pós-Condição:** Registro alterado no banco de dados. **Extensões:** Informações provenientes de outros cadastros.	
Nome do fluxo alternativo (extensão)	Descrição
1. Campos obrigatórios não informados.	1.1. Ator exclui 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 dados deixando-os 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.

Fonte: Autoria própria

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:** Exclusão dos dados cadastrados no sistema. **Evento Iniciador:** Ator solicita exclusão de um registro no sistema. **Atores:** Administrador ou corretor, de acordo com suas funções definidas no caso de uso. **Pré-condição:** Registro estar incluso no sistema. **Sequência de Eventos:** 1. Ator acessa a tela para exclusão do registro. 2. O sistema exclui as informações no banco de dados e informa ao usuário o *status* do procedimento. **Pós-Condição:** Registro excluído no banco de dados. **Extensões:** Registro possuir vínculo com outros cadastros.	
Nome do fluxo alternativo (extensão)	Descrição
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 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:** Consulta dos dados cadastrados no sistema. **Evento Iniciador:** Ator solicita consulta de um registro. **Atores:** Cliente ou corretor, de acordo com suas funções definidas no caso de uso. **Pré-condição:** Registro estar incluso no sistema. **Sequência de Eventos:** 1. Ator acessa a tela para visualização dos dados do registro. 2. O ator indica os filtros desejados para consulta. 3. O sistema apresenta os dados da consulta ao usuário. **Pós-Condição:** Dados da consulta apresentados ao usuário.

Fonte: Autoria própria

No Quadro 8 apresenta a expansão do caso de uso para emissão de relatórios dos seguros e das parcelas a vencer.

Quadro 8 - Gerar relatórios

Caso de uso: Emitir relatórios. **Descrição:** Listagem de seguros ou parcelas que estão prestes a vencer, listagem de seguros por cliente e por tipo de seguro. **Evento Iniciador:** Selecionar a opção de listagem desejada. **Atores:** Corretor. **Pré-condição:** Registros de seguros e de pagamentos cadastrados no sistema. **Sequência de eventos:** 1. Ator seleciona a respectiva opção de listagem. 2. Sistema exibe a tela com os campos da opção selecionada e as opções de exibir, baixar e enviar por e-mail o relatório. 3. Ator preenche os campos e seleciona a opção desejada. 4. Sistema mostra o relatório correspondente. **Pós-condição:** Não há.

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 troca as datas para ficar menor para maior.

Fonte: Autoria própria

A expansão do caso de uso personalizar notificação é apresentada no Quadro 9.

Quadro 9 - Personalizar notificação

Caso de uso: Personalizar notificação. **Descrição:** Configuração da mensagem a ser enviada aos clientes quando seu seguro ou parcela estiver prestes a vencer e dos prazos de quantos dias antes do vencimento enviar a mensagem. **Evento Iniciador:** Selecionar a opção de personalizar notificação. **Atores:** Corretor. **Pré-condição:** Registros de seguros cadastrados no sistema. **Sequência de eventos:** 1. Ator seleciona a respectiva opção de notificação (seguro ou parcela). 2. Sistema exibe o *modal* com o título e texto do *e-mail* e os dois prazos. 3. Ator preenche os campos e salva. 4. Sistema fecha o *modal*. **Pós-condição:** Não há.

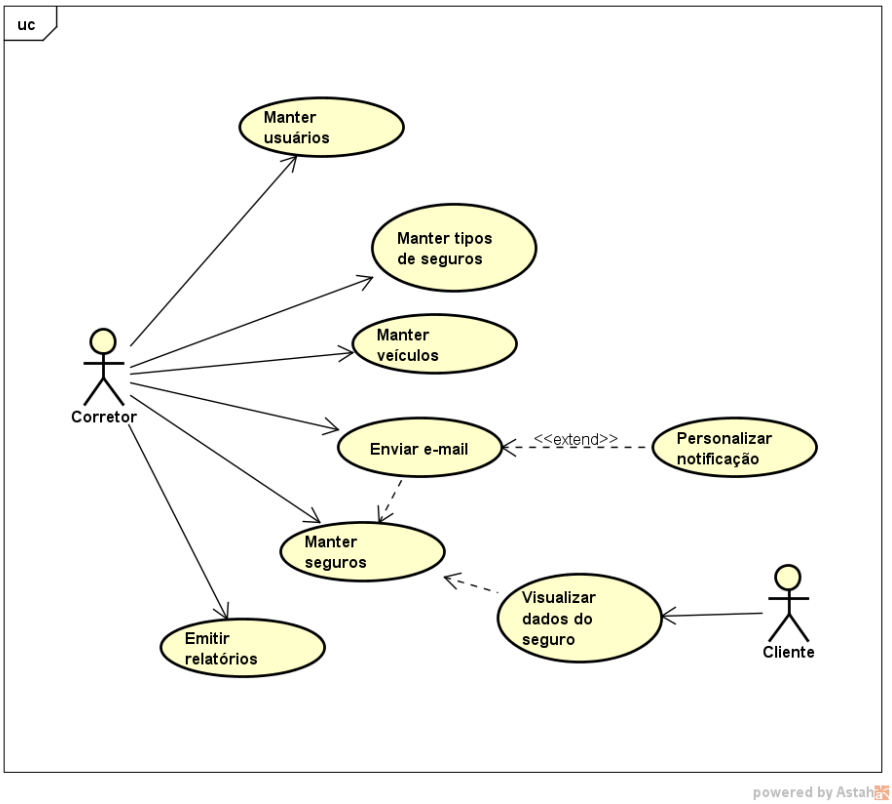
Nome do fluxo alternativo (extensão)	Descrição
1. Ator salvou mensagem em branco ou esqueceu o formato	1.1. Ator excluir o texto padrão ou salvou em branco 1.2. O sistema fornece um botão no *modal* para restaurar o texto padrão.

Fonte: Autoria própria

A Figura 1 apresenta o digrama de casos de uso. Esse diagrama apresenta as funcionalidades essenciais do sistema que são realizadas por seus atores representados pelo corretor e cliente. O corretor tem acesso a todas as operações do sistema, como, por exemplo, manter cadastros e emitir relatórios de pagamentos das parcelas e dos seguros. O cliente tem acesso, apenas, à consulta dos dados dos pagamentos dos seus seguros.

O caso de uso Personalizar Notificação refere-se ao *e-mail* que o corretor pode enviar ao cliente notificando-o da data do pagamento da parcela ou do seguro. A mensagem e o título da mensagem vêm, por padrão, com o texto definido. Contudo, é possível que o corretor altere as informações conforme a necessidade.

Figura 1 – Casos de uso

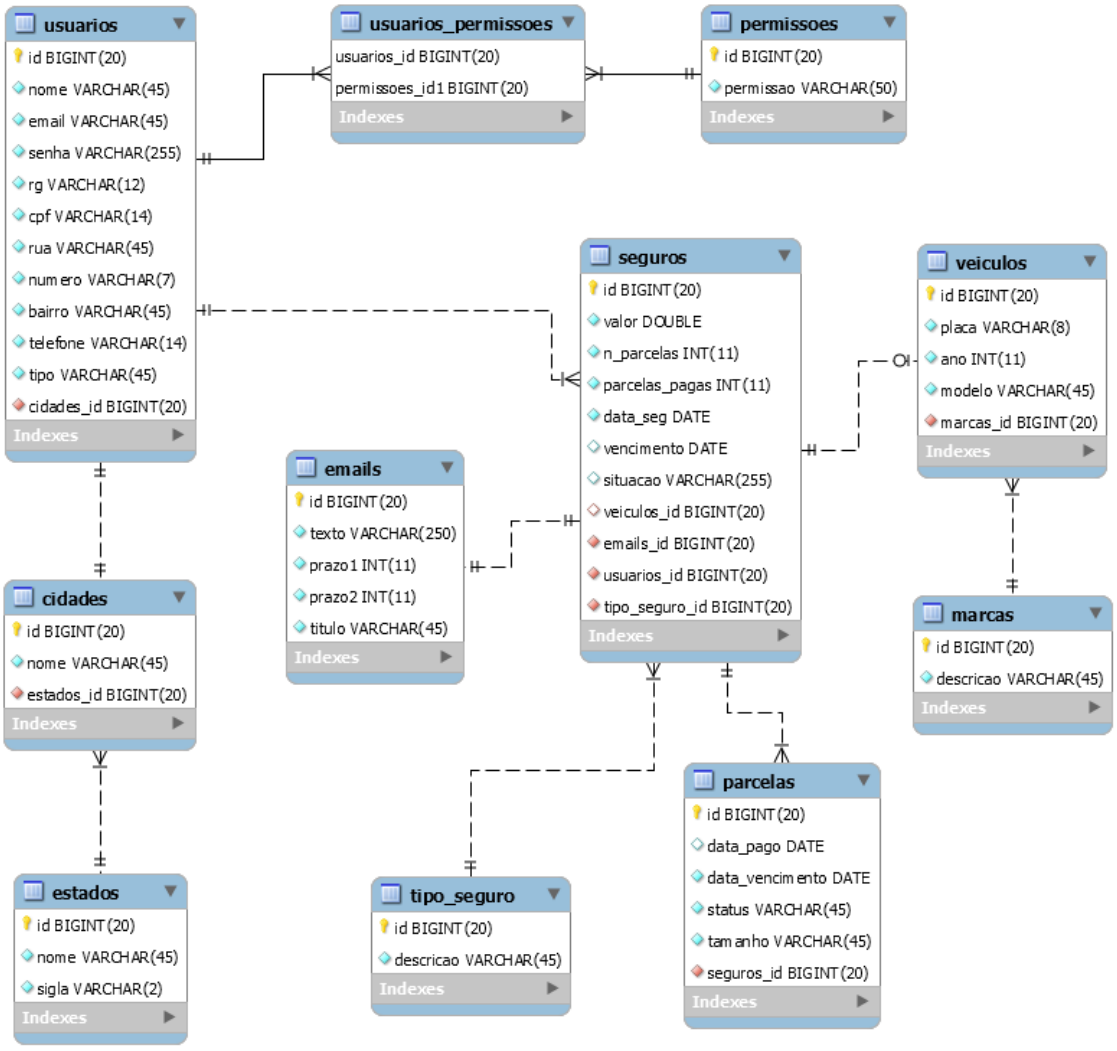


Fonte: Autoria própria

A Figura 2 apresenta o diagrama com as entidades e os relacionamentos representados no banco de dados.

As tabelas principais do sistema são a de usuários e a de seguros. A tabela de usuários armazena os dados de todos os usuários do sistema, que pode ser corretor ou cliente. O tipo de usuário é selecionado no momento em que o cadastro é realizado e é predefinido em tempo de projeto. A tabela de seguros armazena os dados do seguro, como, data de efetivação do seguro, número de parcelas, cliente, tipo de seguro, entre outros. Esses dados serão utilizados para o gerenciamento e o controle dos pagamentos das parcelas dos seguros e do seguro, possibilitando que o corretor possa controlar de forma mais efetiva os vencimentos.

Figura 2 - Diagrama de entidades e relacionamentos do banco de dados



Fonte: Autoria própria

4.3 APRESENTAÇÃO DO SISTEMA

Para ter acesso ao sistema é necessário realizar a autenticação por meio da tela de autenticação, apresentada na Figura 3, na qual há os campos de *e-mail* e senha para serem preenchidos com os dados do usuário.

Figura 3 - Tela de login

Fonte: Autoria própria

Assim que a autenticação for realizada, caso o usuário seja um corretor, ele é redirecionado para o a tela principal que é a de *dashboard* (exibe de forma visual as informações mais importantes do sistema) da Figura 4. Essa tela contém três quadros que apresentam as informações do total de seguros efetivados, número de parcelas pagas e o número de parcelas a pagar e seus respectivos valores calculados. Apresenta, ainda, as informações de seguros efetivados no último ano e o percentual de cada tipo de seguro efetivado em forma de gráficos.

Figura 4 - Tela Principal

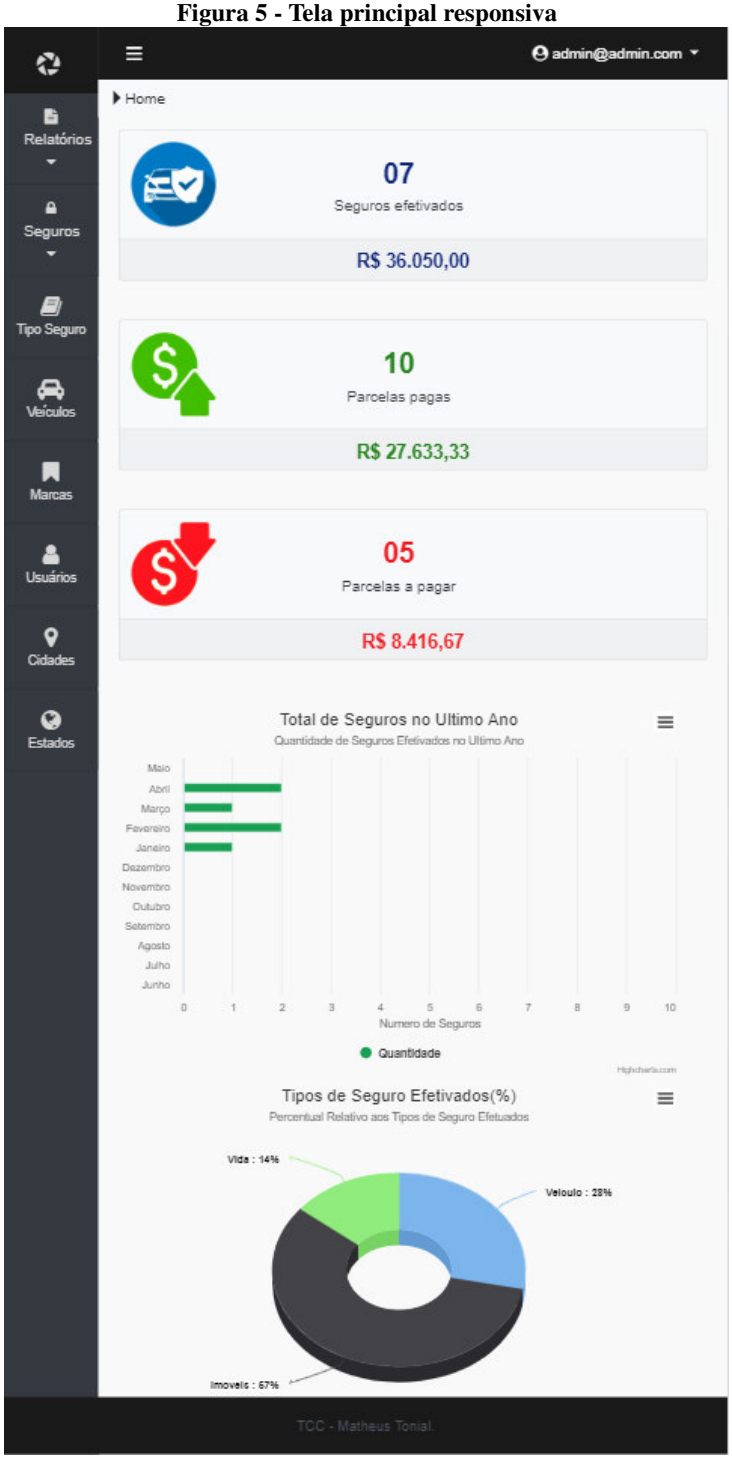
Fonte: Autoria própria

Todas as telas do sistema, após o usuário realizar a autenticação, são compostas pela estrutura de um leiaute padrão, fragmentado em elementos do tipo *header*, *footer*, *sidebar* e o *body* que representam, respectivamente, cabeçalho, rodapé, menu e conteúdo principal da aplicação.

O *header* contém espaço para os dados que representam a identidade visual de uma corretora, os dados do usuário autenticado com um botão *dropdown* que possui as opções de retornar a tela principal e de efetuar o leiaute e um botão para alterar o leiaute do menu. O *sidebar* (barra lateral) possui um menu com os *links* para as telas de relatório e para as telas de listagem e cadastro. Por fim, o conteúdo principal é na parte central onde todas as telas do sistema são apresentadas, sendo a única parte do leiaute que sofre alterações de acordo com a página que o usuário acessar.

A Figura 5 mostra a responsividade do sistema, apresentando o leiaute da tela principal ao acessá-lo por *tablet* ou *smartphone*.

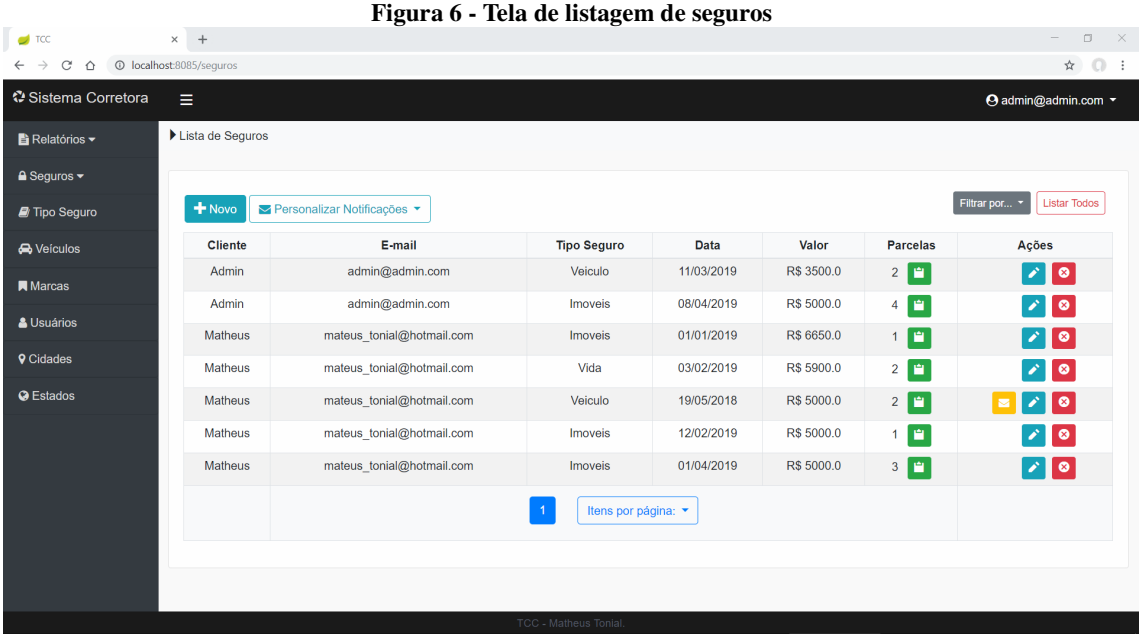
Figura 5 - Tela principal responsiva



Fonte: Autoria própria

A Figura 6 apresenta a tela de listagem de seguros que é acessada por meio do *sidebar* selecionando, no botão *dropdown* de seguros, a opção para exibir todos os seguros, sendo o conteúdo exibido no corpo do *layout* padrão.

Figura 6 - Tela de listagem de seguros



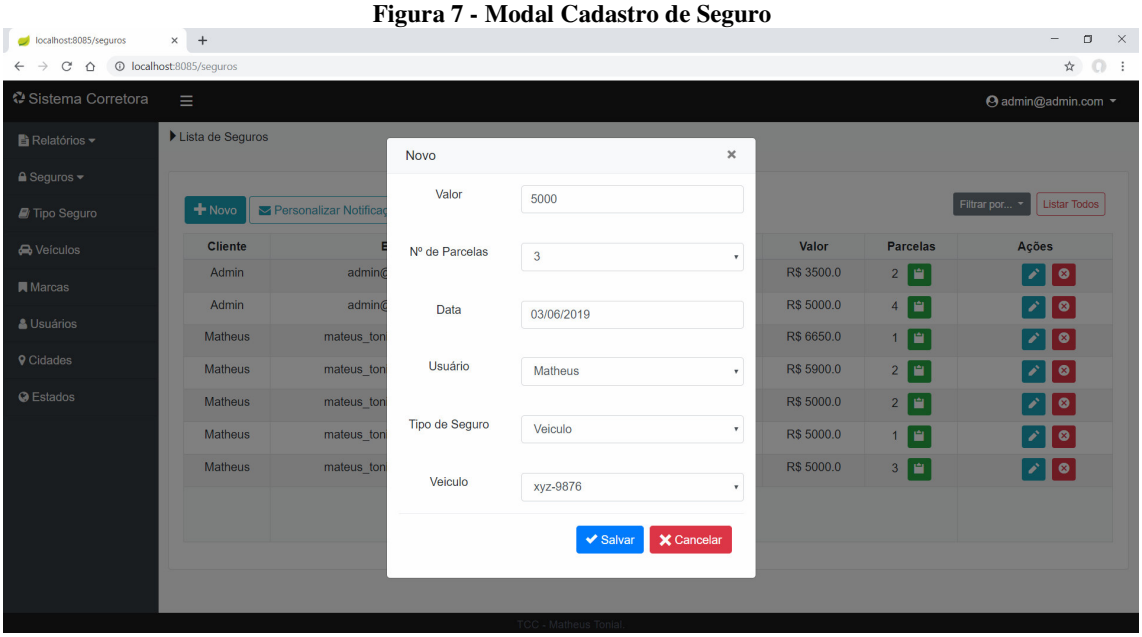
Fonte: Autoria própria

A tela de listagem de seguros da Figura 6 consiste de uma tabela listando todos os seguros registrados. Na parte superior esquerda possui os botões para cadastrar um novo seguro e um botão *dropdown* para personalizar as mensagens de *e-mail* que serão enviadas quando um seguro ou parcela estiver próxima da data do vencimento, à direita também há um botão *dropdown* com as opções de filtros para exibir os seguros agrupados por cliente, tipo de seguro e data e o botão para listar todos os seguros.

Abaixo desses botões são exibidas informações dos seguros registrados, como, nome e e-mail do cliente, tipo do seguro, data de efetivação, valor total, número de parcelas e um botão para visualizar as informações de cada parcela. Também há uma coluna de ações com botões para editar e excluir um seguro e um botão para enviar um *e-mail* notificando o cliente de que o seguro ou parcela irá vencer. Esse botão é visível somente para os seguros que estão com o prazo de pagamento próximos da data do vencimento. Após a tabela de listagem há a paginação utilizada para dividir a listagem de seguros em páginas e um botão *dropdown* com a opção de selecionar a quantidade de registros que serão apresentados por página.

Ao clicar no botão “Novo”, para cadastrar um novo seguro, uma tela *modal* será exibida, como apresentado na Figura 7, com um formulário que deverá ser preenchido para o cadastro do seguro.

Figura 7 - Modal Cadastro de Seguro

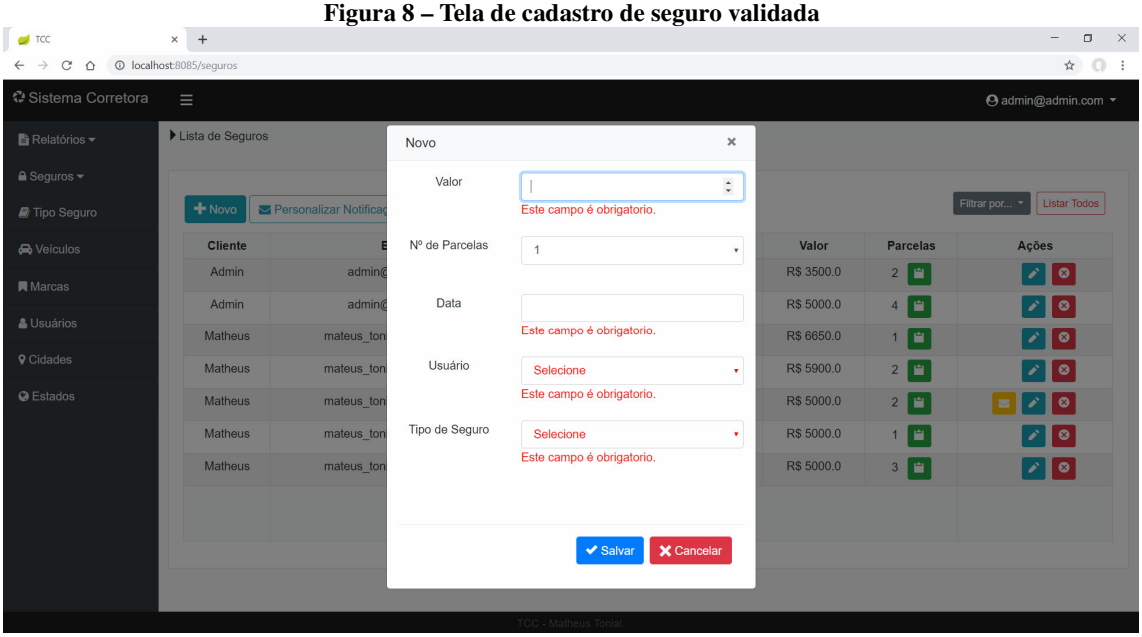


Fonte: Autoria própria

No formulário da Figura 7 estarão disponíveis os seguintes campos: valor do seguro; número de parcelas, data de efetivação (previamente preenchido com a data atual, mas permite alteração), seleção de usuário para o qual o seguro será cadastrado e seleção do tipo de seguro. Caso seja selecionada a opção de veículo, será exibido o campo para selecionar o veículo (que já estará cadastrado).

Caso o usuário clicar na opção “Salvar” e o formulário não esteja preenchido corretamente, o sistema irá efetuar a validação dos campos e alertar sobre os campos com informação incorreta, como visto na Figura 8.

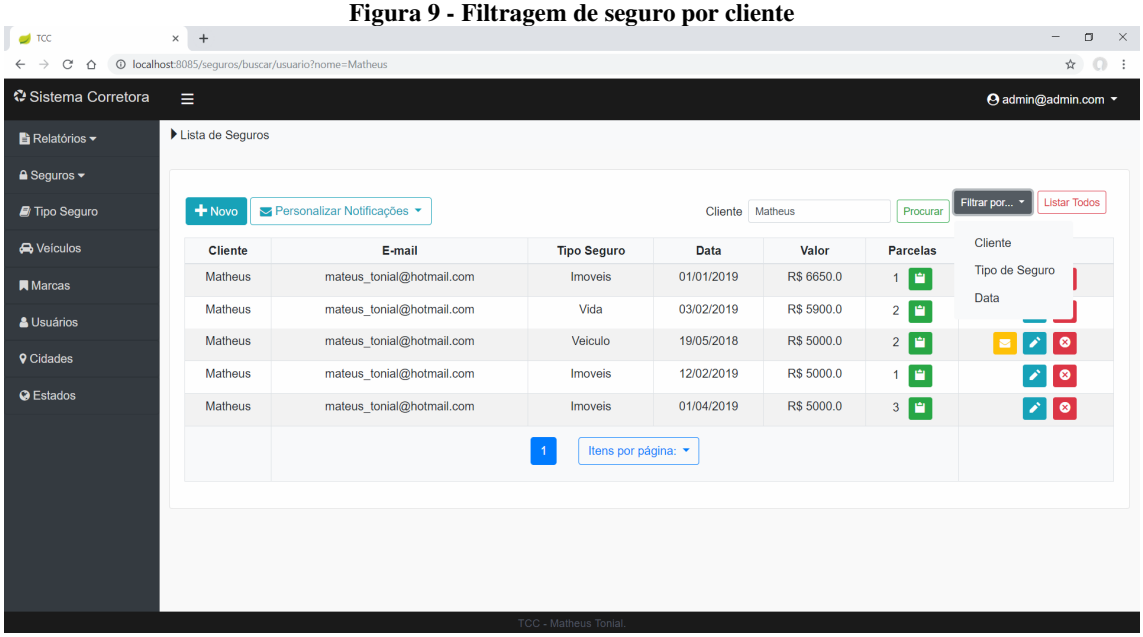
Figura 8 – Tela de cadastro de seguro validada



Fonte: Autoria própria

Nas consultas de seguros, quando o usuário selecionar umas das opções de seleção (cliente, tipo de seguro ou data), ficará disponível um campo para ser preenchido e, ao clicar no botão “Procurar” será realizada a filtragem dos dados conforme a opção escolhida e o dado informado. A Figura 9 exemplifica a filtragem dos dados pelo cliente.

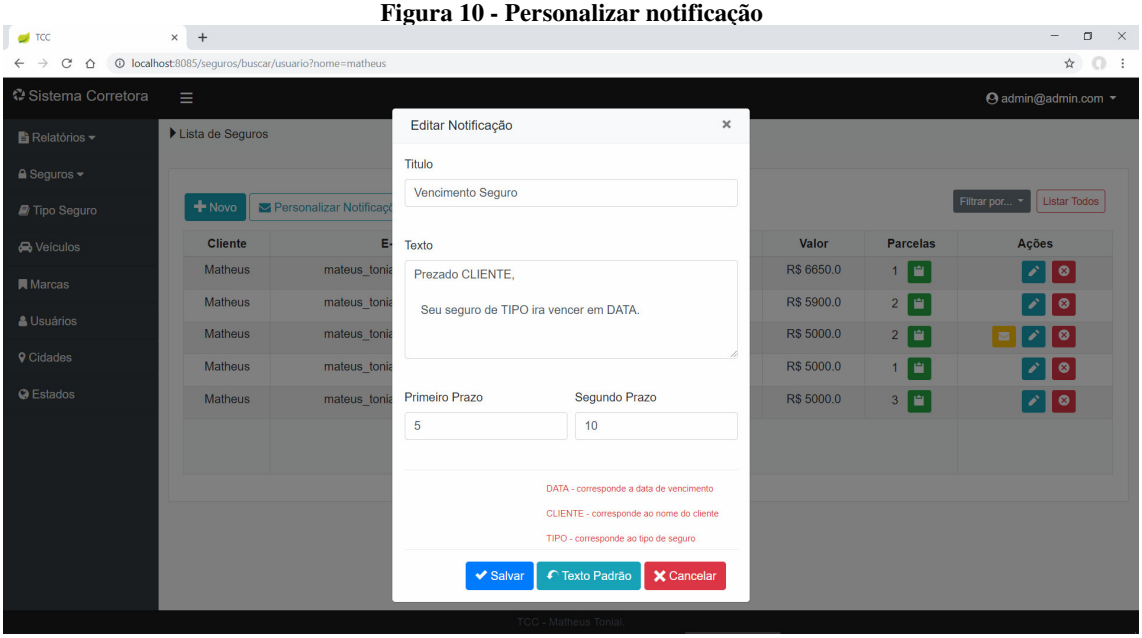
Figura 9 - Filtragem de seguro por cliente



Fonte: Autoria própria

O botão de “Personalizar Notificações” permite que o administrador personalize o *e-mail* que será enviado para o cliente, notificando-o da data do vencimento da parcela ou seguro. Ao selecionar uma das opções (parcela ou seguro), uma janela *modal,* apresentada na Figura 10, será exibida, sendo possível editar a notificação a ser enviada por *e-mail* aos clientes. Nessa tela há os campos de título e texto que compõem o *e-mail* e os prazos que correspondem a quantos dias antes do vencimento a notificação ficará disponível. O botão com o valor texto padrão restaura o texto original.

Figura 10 - Personalizar notificação



Fonte: Autoria própria

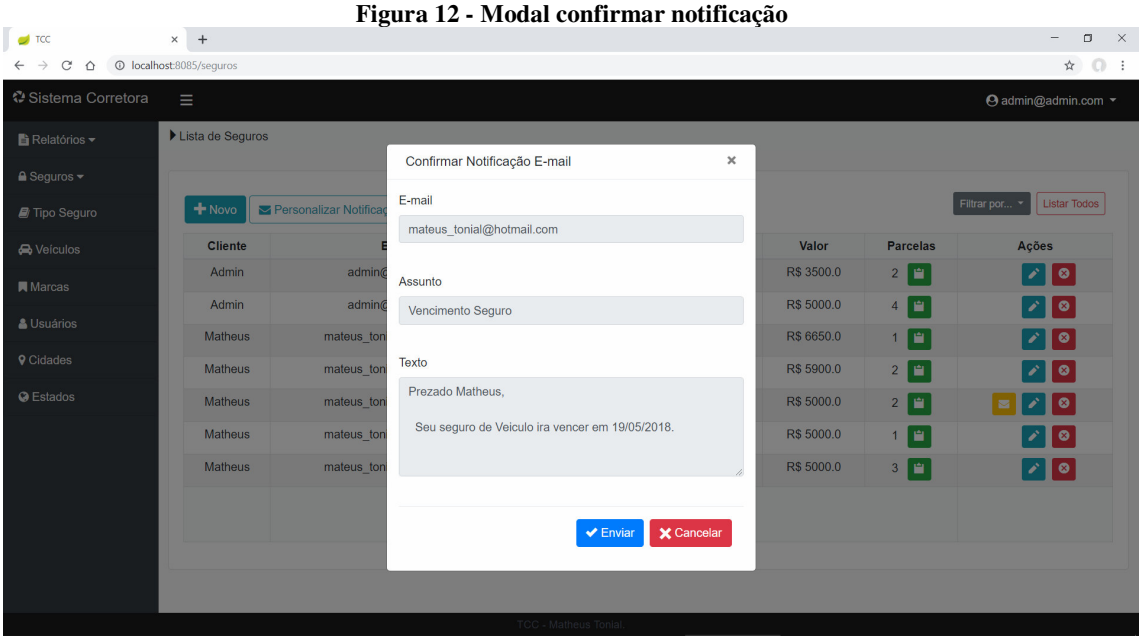
Na coluna de parcelas, da tela de listagem de seguros, há um botão para visualizar as parcelas específicas de cada seguro. Ao clicar nesse botão será exibida uma janela *modal* com a tabela de parcelas, como pode ser observado na Figura 11, detalhando o nome do cliente, o tipo de seguro, o vencimento de cada parcela, o número correspondente dessa parcela, a data de pagamento (caso já tenha sido efetuado) e a situação da parcela (pago ou aberto) e o botão para confirmar o pagamento que, ao ser clicado muda o *status*, caso o pagamento da parcela tenha sido efetuado, a situação muda para “pago” e o campo para a data do pagamento é preenchido com a data atual e, caso o pagamento não foi efetuado, a situação mudará para “aberta” e a data de pagamento fica em branco.

Figura 11 - Modal parcelas do seguro

Fonte: Autoria própria

Ainda na tabela de listagem de seguros, da Figura 11, na coluna de ações, está disponível aos seguros e parcelas que estão próximos da data do vencimento, o botão de enviar notificação que abre a janela *modal* com a mensagem padrão, exibida na Figura 12 para que o corretor possa enviar o *e-mail* ao cliente.

Figura 12 - Modal confirmar notificação



Fonte: Autoria própria

As telas que possuem as operações de incluir excluir, consultar e pesquisar seguem o mesmo padrão de leiaute e de funcionalidades genéricas, exceto se houver funções específicas, como as apresentadas na tela de seguros.

O sistema também permite que sejam gerados relatórios relacionados aos seguros a vencer, agrupados por cliente, parcelas a vencer, parcelas vencidas e por tipo de seguro. Todas as telas de relatórios também possuem uma estrutura padronizada. No cabeçalho de cada relatório são exibidos os campos correspondentes a cada relatório. A Figura 13 exemplifica a tela de seguros a vencer, na qual são exibidos os campos de data inicial, que vem com a data atual preenchida e a data final e os botões padrões para gerar o relatório, realizar o *download* no formato *Portable Document Format* (PDF) e enviá-lo por *e-mail*.

Figura 13 - Tela de relatório

Fonte: Autoria própria

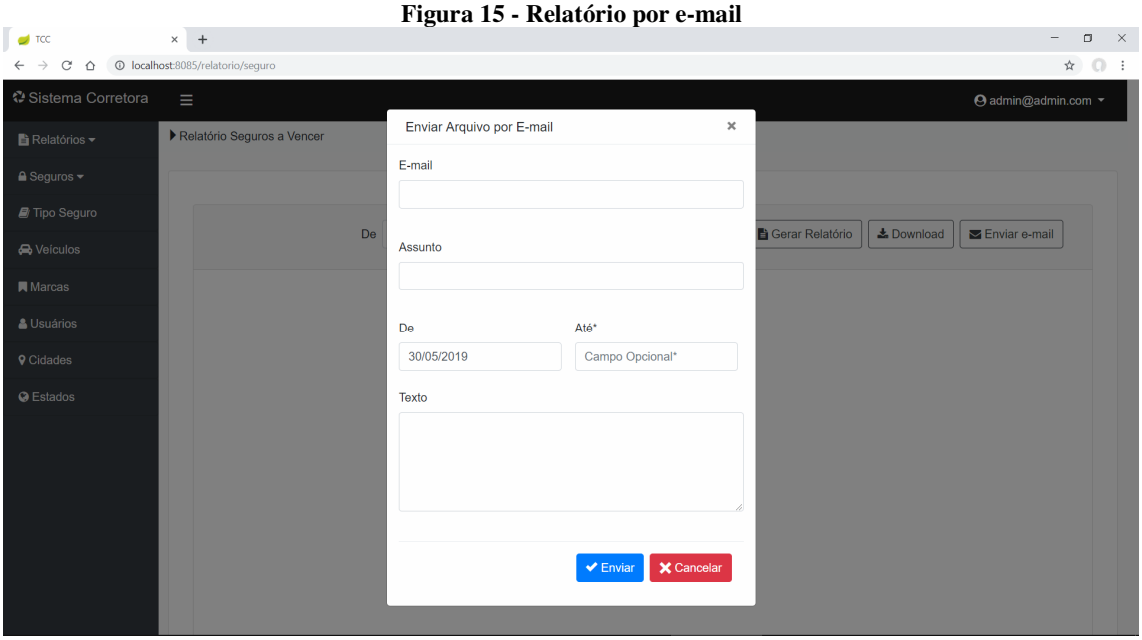
Ao gerar um relatório, o arquivo é apresentado na tela conforme exemplifica a Figura 14. Caso não tenha sido informada a data final é gerado o relatório de um ano a partir da data inicial.

Figura 14 - Relatório de seguros a vencer

Fonte: Autoria própria

Quando clicar na opção de enviar o relatório por *e-mail*, uma janela *modal* é aberta, devendo ser informados os campos de endereço de *e-mail*, assunto, texto e as datas de início e fim do relatório, conforme exemplifica a Figura 15.

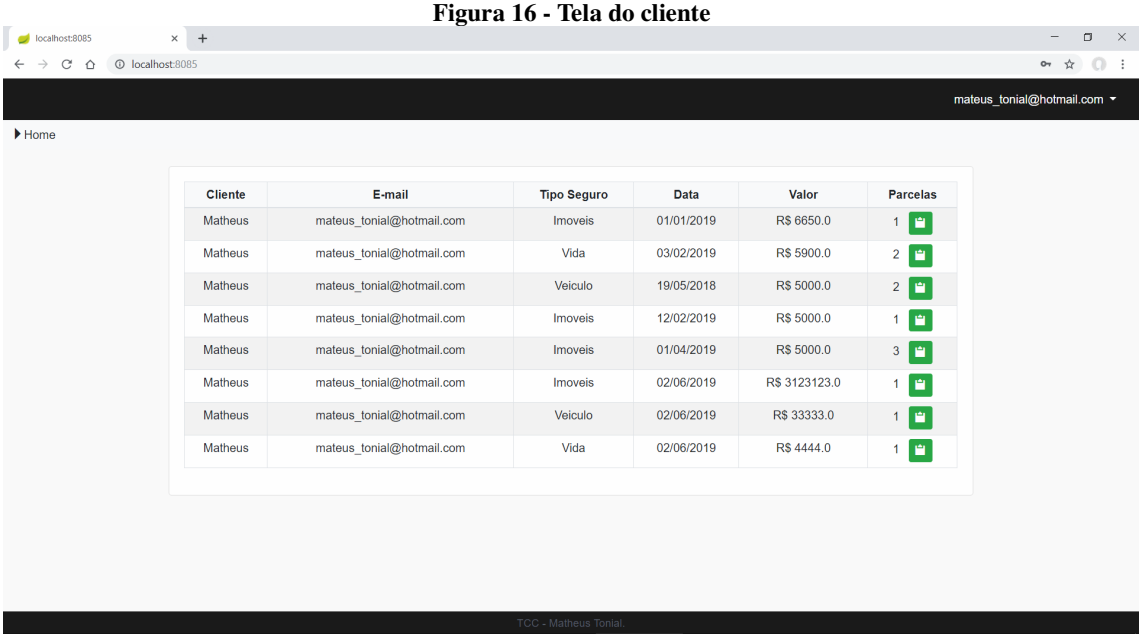
Figura 15 - Relatório por e-mail



Fonte: Autoria própria

Após o cliente autenticar-se no sistema, será exibida a tela da Figura 16, que é uma listagem dos seus seguros efetivados. Assim, o cliente pode acompanhar os pagamentos realizados, as datas de vencimento das parcelas e dos seguros.

Figura 16 - Tela do cliente



Fonte: Autoria própria

4.4 IMPLEMENTAÇÃO DO SISTEMA

O Thymeleaf pode incluir partes de outras páginas como fragmentos usando o comando *th:replace*, que permite o agrupamento de fragmentos em uma ou várias páginas. Foi implementado o conceito de leiaute padrão por meio desse recurso, como mostrado na Listagem 1, que exibe a página *layout.html*. Essa página acessa os conteúdos necessários como, o menu *sidebar*, o *header* e o *footer*, que foram estruturados individualmente em arquivos HTML, e os integra junto ao corpo e apresenta ao usuário como uma página só. Nas linhas 27, 33 e 40 da Listagem 1, o comando th:replace substitui a *div* pela sua correspondente em outra página HTML. A expressão após os dois pontos duplos é um seletor de fragmento (nome do fragmento).

Para exibir o conteúdo principal (corpo), que é um componente que é alterado de acordo com a página que o usuário está acessando e, como essas páginas dependem de uma conexão com o *controller*, é utilizado o comando “layout:fragment” pois ele oferece recursos mais avançados para trabalhar com as respostas oriundas do *controller.*

Listagem 1 - Layout Padrão

```
1   <!DOCTYPE html>
2   <html lang="pt" xmlns="http://www.w3.org/1999/xhtml"
3         xmlns:th="http://www.thymeleaf.org"
4         xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout"
5         xmlns:sec="http://www.thymeleaf.org/extras/spring-security">
6   <head>...</head>
22  <body>
23  <div class="layout-main">
24      <aside>
25          <div class="wrapper" sec:authorize="hasRole('ROLE_ADMIN')">
26              <div class="row row-offcanvas row-offcanvas-left"
27                   th:replace="fragments/sidebar :: nav-sidebar">
28                  <span>menu</span>
29              </div>
30          </div>
31      </aside>
32      <div class="container-fluid ">
33          <header th:replace="fragments/header :: cabecalho">
34              <div>header</div>
35          </header>
36          <section layout:fragment="corpo">
37              <div>conteudo</div>
38          </section>
39      </div>
40      <footer th:replace="fragments/footer :: rodape">
41          <div>footer</div>
42      </footer>
43  </div>
44  <script>...</script>
87  </body>
88  </html>
```

Fonte: Autoria própria

Para que o sistema saiba por quais componentes devem ser feitos as substituições, é preciso utilizar outras funcionalidades do Thymeleaf nas páginas correspondentes, como mostra no exemplo do arquivo footer.html da Listagem 2. Na linha 8, é utilizado o comando t*h:fragment* para distinguir que é aquele segmento do código que irá substituir o campo apontado na linha 40 da Listagem 1. O mesmo procedimento é utilizado nas páginas header.html e sidebar.html para substituir seus respectivos campos do leiaute padrão.

Outra funcionalidade que se destaca na Listagem 1 é o comando da linha 19 *sec:authorize="hasRole('ROLE_ADMIN')"*, que é utilizado para restringir o acesso do menu principal apenas aos usuários com papel de administrador isso é, apenas aos corretores.

Listagem 2 - footer.html

```
1   <!DOCTYPE html>
2   <html lang="pt" xmlns="http://www.w3.org/1999/xhtml"
3         xmlns:th="http://www.thymeleaf.org">
4   <head>
5       <meta charset="UTF-8">
```

```
6    </head>
7    <body>
8    <footer class="layout-footer" th:fragment="rodape">
9        <div class="container">
10           <span class="footer-copy">TCC - Matheus Tonial.</span>
11       </div>
12   </footer>
13   </body>
14   </html>
```

Fonte: Autoria própria

A restrição de acesso também é usada na página principal, como pode ser visto na Listagem 3, que contém um segmento da página *index.html*, para diferenciar o conteúdo disponível ao corretor e ao cliente, linhas 24 e 232, respectivamente, separando-os por meio do comando *sec:authorize* que torna a *div* disponível somente a quem tem a permissão adequada (administrador ou cliente).

Listagem 3 - index.html

```
1    <!DOCTYPE html>
2    <html lang="pt" xmlns="http://www.w3.org/1999/xhtml"
3          xmlns:th="http://www.thymeleaf.org"
4          xmlns:sec="http://www.thymeleaf.org/extras/spring-security"
5          xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout4"
6          layout:decorate="~{layout}">
...
11   <body>
12   <section class="layout-content " layout:fragment="corpo">
...
24       <div class="container" sec:authorize="hasRole('ADMIN')">...</div>

232      <div class="container" sec:authorize="hasRole('USER')">...</div>
...
299  </section>
300  </body>
301  </html>
```

Fonte: Autoria própria

Ainda na Listagem 3, destaca-se a linha 12 na qual é utilizado o comando *layout:fragment* para especificar a seção que será apresentada dentro do corpo na Listagem 1 e para efetuar a ligação entre as duas páginas é utilizado o componente *decorate,* na linha 6, que indica qual página contém o campo a ser substituído que, nesse caso, é o leiaute.

A Listagem 4 mostra como foi realizada a configuração do Spring Security. Na linha 19 é utilizada a anotação *@EnableWebSecurity* para indicar ao Spring que essa é a classe responsável pela segurança e habilitar o seu suporte *web* e é estendida a classe *WebSecurityConfigurerAdapter,* na linha 21, para ter acesso ao seus métodos.

O método *configure* foi utilizado em três lugares da classe: na linha 34 para definir quem tem autorização de acesso às páginas do sistema, na linha 56 para determinar os caminhos do projeto a serem ignorados pelo sistema e na linha 78 para efetuar a autenticação do usuário que está tentando realizar o *login*. Para isso, é utilizado o *UserDetailsService* na linha 68, para pegar os dados do usuário no banco de dados por meio do *e-mail* fornecido na tela de autenticação e, então, com o *PasswordEncoder* na linha 73, criptografar e comparar a senha utilizada no *login* com a senha registrada no banco de dados.

Listagem 4 – WebSecurityConfig

```
1   package utfpr.edu.br.tcc.config;
2
3   import ...
18
19  @EnableWebSecurity
20  @EnableGlobalMethodSecurity(securedEnabled = true)
21  public class WebSecurityConfig extends WebSecurityConfigurerAdapter {
22
23      @Autowired
24      private UsuarioRepository usuarioRepository;
25
26      @Autowired
27      private UsuarioService usuarioService;
28
29      public WebSecurityConfig(UsuarioService usuarioService){
30          this.usuarioService = usuarioService;
31      }
32
33      @Override
34      protected void configure(HttpSecurity http) throws Exception{
35          http.csrf().disable().exceptionHandling().accessDeniedPage("/403")
36                  .and().formLogin().loginPage("/login").defaultSuccessUrl("/")
37                  .failureUrl("/login?error=bad_credentials").permitAll()
38                  .and().authorizeRequests()
39                  .antMatchers("/usuarios/**").permitAll()
40                  .antMatchers("/index/**").hasAnyRole("USER", "ADMIN")
41                  .antMatchers("/estados/**").hasRole("ADMIN")
42                  .antMatchers("/cidades/**").hasRole("ADMIN")
43                  .antMatchers("/emails/**").hasRole("ADMIN")
44                  .antMatchers("/marcas/**").hasRole("ADMIN")
45                  .antMatchers("/veiculos/**").hasRole("ADMIN")
46                  .antMatchers("/relatorios/**").hasRole("ADMIN")
47                  .antMatchers("/tipo_Seguros/**").hasRole("ADMIN")
48                  .antMatchers("/seguros/**").hasRole("ADMIN")
49                  .antMatchers("/relatorio/**").hasRole("ADMIN")
50                  .antMatchers("/**").hasAnyRole("USER", "ADMIN");
51
52          http.headers().frameOptions().sameOrigin();
53      }
54
55      @Override
56      public void configure(WebSecurity web) throws Exception{
57          web.ignoring().antMatchers("/fragments/**")
58                      .antMatchers("/images/**")
59                      .antMatchers("/webjars/**")
60                      .antMatchers("/assets/**")
```

```
61                  .antMatchers("/css/**")
62                  .antMatchers("/static/**")
63                  .antMatchers("../resources/static/js/**")
64                      .antMatchers("/layout/**");
65      }
66
67      @Bean
68      public UserDetailsService userDetailsService(){
69          return new UsuarioService(usuarioRepository);
70      }
71
72      @Bean
73      public PasswordEncoder passwordEncoder(){
74          return new BCryptPasswordEncoder(10);
75      }
76
77      @Override
78      protected void configure(AuthenticationManagerBuilder auth)
  throws Exception{
79          auth.userDetailsService(userDetailsService())
  .passwordEncoder(passwordEncoder());
80      }
81
82      @Bean
83      DaoAuthenticationProvider authenticationProvider(){
84          DaoAuthenticationProvider daoAuthenticationProvider =
  new DaoAuthenticationProvider();
85          daoAuthenticationProvider.setPasswordEncoder(passwordEncoder());
86          daoAuthenticationProvider.setUserDetailsService(this.usuarioService);
87          return daoAuthenticationProvider;
88      }
89  }
```

Fonte: Autoria própria

A Listagem 5 apresenta como foi desenvolvida a classe de seguro. Utilizou-se a anotação @*Entity*, na linha 13, para vincular tabela do banco de dados e a @*Table* para definir o nome da tabela correspondente, tornando assim todos os atributos da classe uma coluna na tabela do banco. Há campos na tabela que não podem ser nulos e, para isso, foi usado a anotação @*NotNull* nos atributos correspondentes.

Listagem 5 - Classe Seguro

```
1   package utfpr.edu.br.tcc.model;
2
3   import ...
12
13  @Entity
14  @Table(name = "seguros")
15  @Data
16  @AllArgsConstructor
17  @NoArgsConstructor
18  @EqualsAndHashCode
19  @ToString
20  public class Seguro {
21
22      @Id
```

```
23      @GeneratedValue(strategy = GenerationType.IDENTITY)
24      private Long id;
25
26      @Column(nullable = false)
27      private Double valor;
28
29      @NotNull
30      @Column(nullable = false)
31      private Integer nParcelas;
32
33      @NotNull
34      @Column(nullable = false)
35      private Integer parcelasPagas;
36
37      @Column(nullable = true)
38      private String situacao;
39
40      @DateTimeFormat(pattern = "dd/MM/yyyy")
41      @Column(name= "dataSeg", nullable = false,
  columnDefinition = "DATE")
42      private LocalDate dataSeg;
43
44      @DateTimeFormat(pattern = "dd/MM/yyyy")
45      @Column(name= "vencimento", nullable = true,
  columnDefinition = "DATE")
46      private LocalDate vencimento;
47
48      @NotNull
49      @ManyToOne
50      @JoinColumn(name = "usuario_id", referencedColumnName = "id",
  nullable = false)
51      private Usuario usuario;
52
53      @NotNull
54      @ManyToOne
55      @JoinColumn(name = "tipo_id", referencedColumnName = "id",
  nullable = false)
56      private Tipo_Seguro tipoSeguro;
57
58      @OneToOne
59      @JoinColumn(name = "veiculo_id", referencedColumnName = "id",
  nullable = true, columnDefinition = "bigint")
60      private Veiculo veiculo;
61
62      @NotNull
63      @OneToOne
64      @JoinColumn(name = "email_id", referencedColumnName = "id",
  nullable = false)
65      private Email email;
66  }
```

Fonte: Autoria própria

A Listagem 6 contém uma parte do código do *controller* de seguro, mais especificamente os métodos de salvar e listar seguros. Destaca-se na linha 46 o caminho que se encontra o HTML que fará as requisições e receberá os dados do *controller*.

Na linha 55 é definido o método para salvar os dados do seguro, o qual recebe os dados do formulário do seguro e define os valores para os outros atributos que não estão presentes na tela de cadastro. Com a mensagem de *e-mail*, o número de parcelas pagas e a data de vencimento do seguro, ainda dentro do método após salvar o seguro na linha 59 é realizado um laço de repetição para criar cada entidade de parcela correspondente a quantidade de parcelas selecionadas pelo usuário.

O método para listar os registros do seguro é exibido na linha 94. Esse método, além de efetuar a busca pelos seguros, também chama o método carregar, da linha 77, que faz a busca pelas parcelas e *e-mails* em suas respectivas classes. Ainda no método é feito uma lista das parcelas contendo a data de vencimento, o *status* e o *id* do seguro. Essa lista é utilizada para analisar o prazo de vencimento das parcelas e apresentar o botão para enviar a notificação.

Listagem 6 - SeguroController

```
1   package utfpr.edu.br.tcc.controller;
2
3   import ...
26
27  @Controller
28  @RequestMapping("/seguros")
29  public class SeguroController {
30
31      @Autowired
32      private SeguroRepository repository;
33      @Autowired
34      private UsuarioRepository usuarioRepository;
35      @Autowired
36      private Tipo_SeguroRepository tipo_seguroRepository;
37      @Autowired
38      private VeiculoRepository veiculoRepository;
39      @Autowired
40      private EmailRepository emailRepository;
41      @Autowired
42      private ParcelasRepository parcelasRepository;
43      @Autowired
44      EnvioEmail envioEmail;
45
46      ModelAndView modelAndView = new ModelAndView("seguro/list");
53
54      @PostMapping("/salvar")
55      public ModelAndView salvar(Seguro seguro, BindingResult result,
  RedirectAttributes attr) {
56          seguro.setEmail(emailRepository.findOne(1L));
57          seguro.setParcelasPagas(0);
58          seguro.setVencimento(seguro.getDataSeg().plusYears(1L));
59          repository.save(seguro);
60          int nPar = 1;
61          do {
62              Parcelas parcelas = new Parcelas();
63              parcelas.setDataVencimento(seguro.getDataSeg()
```

```
  .plusMonths(nPar));
64                parcelas.setDataPago(null);
65                parcelas.setStatus("aberto");
66                parcelas.setSeguro(seguro);
67                String tamanho = Integer.toString(nPar)+"/"
  +Integer.toString(seguro.getNParcelas());
68                parcelas.setTamanho(tamanho);
69                parcelasRepository.save(parcelas);
70                nPar++;
71            } while (nPar <= seguro.getNParcelas());
72
73            attr.addFlashAttribute("mensagem", "Seguro inserido com
  sucesso!");
74            return new ModelAndView("redirect:/seguros");
75        }
76
77        public ModelAndView carregar(@PageableDefault Pageable pageable){
78            modelAndView.addObject("parcelas",
  parcelasRepository.findAll(pageable));
79            modelAndView.addObject("emails",
  emailRepository.findAllByOrderById(pageable));
80            modelAndView.addObject("email", new Email());
81            List<Parcelas> pc = parcelasRepository.findAllByOrderById();
82
83            StringBuilder list1 = new StringBuilder();
84            for(int i = 1; i < pc.size(); i++){
85                list1.append(pc.get(i-1).getDataVencimento()
  .toString()).append(";")
86                        .append(pc.get(i-1).getStatus()).append(";")
87                        .append(pc.get(i-1).getSeguro()
  .getId()).append(",");
88            }
89            modelAndView.addObject("listaParcelaSeg", list1);
90            return modelAndView;
91        }
92
93        @GetMapping
94        public ModelAndView listar(@PageableDefault Pageable pageable) {
95            Page<Seguro> page = repository.findAllByOrderById(pageable);
96            modelAndView.addObject("seguros", page);
97            modelAndView.addObject("seguro", new Seguro());
98            carregar(pageable);
99            return modelAndView;
100       }
101   }
```

Fonte: Autoria própria

A Listagem 7 representa um segmento do seguro/list.html que é referenciado no *controller* de seguro. Esse segmento corresponde à tabela com a listagem de seguros. Na linha 136 é utilizado o comando *th:each* para preencher as linhas da tabela com os seguros que foram passados pelo método responsável por listar os seguros no *controller*, é utilizada a variável “s” para representar cada objeto dentro da lista de seguros e exibir seus elementos nas respectivas colunas.

Dentre essas colunas é importante ressaltar a linha 145, na qual se encontra o botão para visualizar as parcelas sendo feita uma requisição para a página de parcelas que é redirecionada para um componente *iframe* pelo comando *target*. O botão responsável por enviar a notificação por *e-mail,* na linha 152, que permanece oculto até que o prazo de vencimento do seguro ou das parcelas esteja prestes a esgotar, na mesma coluna há dois campos *labels* que ficam ocultos e são usados pelo JavaScript para tornar o botão visível, quando necessário.

Listagem 7 - Lista de Seguros

```
122 <div class="table-responsive">
123     <table id="tblListaSeguro" class="table table-striped table-bordered
   table-hover table-sm">
124         <thead>
125         <tr class="bg-light degrade" align="center">
126             <th>Cliente</th>
127             <th>E-mail</th>
128             <th>Tipo Seguro</th>
129             <th>Data</th>
130             <th>Valor</th>
131             <th>Parcelas</th>
132             <th>Ações</th>
133         </tr>
134         </thead>
135         <tbody>
136         <tr th:each="s : ${seguros}" align="center">
137             <td th:text="${s.id}" class="idSeguro" hidden></td>
138             <td th:text="${s.usuario.nome}"></td>
139             <td th:text="${s.usuario.email}"></td>
140             <td th:text="${s.tipoSeguro.descricao}"></td>
141             <td th:text="${#temporals.format(s.dataSeg, 'dd/MM/yyyy')}"
   class="dataSeguro"></td>
142             <td th:text=" 'R$ ' + ${s.valor}"></td>
143             <td>
144                 <label th:text="${s.nParcelas}" class="mr-sm-2"></label>
145                 <a class="btn btn-success btn-sm btnTabela" name="parcelas"
146                    th:href="@{/parcelas/seguro/{id} (id=${s.id})}"
147                    target="myIframe" title="Ver Parcelas"
   data-toggle="tooltip">
148                     <span class="oi oi-clipboard" aria-hidden="true"></span>
149                 </a>
150             </td>
151             <td>
152                 <a th:href="@{/seguros/confirmar/{id} (id=${s.id})}"
   name="email"
153                    class="btn btn-warning text-white btn-sm mr-sm-1
   btnNotificacao"
154                    title="Enviar notificação por e-mail"
   data-toggle="tooltip"
155                    style="visibility: hidden">
156                     <span class="fa fa-envelope" aria-hidden="true"></span>
157                 </a>
158                 <label th:text="${s.email.prazo1}" class="prazo1"
   hidden></label>
159                 <label th:text="${s.email.prazo2}" class="prazo2"
   hidden></label>
```

```
160                 <a th:href="@{/seguros/{id} (id=${s.id})}" name="edit"
161                    class="btn btn-info btn-sm btnEditarSeguro mr-sm-1">
162                     <span class="oi oi-pencil" title="Editar"
   aria-hidden="true"></span>
163                 </a>
164                 <input type="hidden" name="_method" value="DELETE"/>
165                 <button class="btn btn-danger btn-sm mr-sm-1" name="delete"
166                     data-toggle="modal" data-target="#confirmRemove"
167                     th:attr="data-id=${s.id}, data-name=${s.usuario.nome}">
168                      <span class="oi oi-circle-x" title="Excluir"
   aria-hidden="true"></span>
169                 </button>
170             </td>
171         </tr>
221         </tbody>
222     </table>
223     <label th:text="${listaParcelaSeg}" id="listagemParcela" hidden></label>
224 </div>
```

Fonte: Autoria própria

A Listagem 8 exibe o código da janela *modal* das parcelas que consiste em um componente *Inline Frame* (*iframe*) que serve para integrar um documento ou até uma página *web* externa dentro da página atual. Nesse caso, ele recebe a requisição e irá apresentar a página com a lista de parcelas, que possui o mesmo conceito da listagem de seguros apresentado anteriormente.

Listagem 8 - Modal de parcelas

```
228 <div th:replace="fragments/modalSeguro :: modalSeguro"></div>
229 <div class="tabelaParcelas">
230     <div class="modal fade" id="tblParcelas" tabindex="-1"
   role="dialog" aria-labelledby="exampleModalLabel"
231          aria-hidden="true">
232         <div class="modal-dialog modal-lg" role="document">
233             <div class="modal-content">
234                 <div class="modal-header">
235                     <h5 class="modal-title" id="exampleModalLabel">
   Parcelas do Seguro</h5>
236                     <button type="button" class="close"
   data-dismiss="modal" aria-label="Close">
237                         <span aria-hidden="true"></span>
238                     </button>
239                 </div>
240                 <div class="modal-body">
241                     <div>
242                         <iframe name="myIframe" id="myIframe"
   frameborder="0" scrolling="no" src=""
243                                 style="width: 635px; height: 265px;
   border: none; margin: 0 auto; display: block;">parcelas...
244                         </iframe>
245                     </div>
246                 </div>
247             </div>
248         </div>
249     </div>
```

```
250 </div>
```

Fonte: Autoria própria

A Listagem 9 exibe como foi desenvolvida a validação dos campos do formulário de cadastro de seguros. Foi utilizada a biblioteca JavaScript, *jQuery* e o *plugin validation* que fornece um conjunto de funções de validação. A função inicia na linha 206, sendo passado o valor do *id* do formulário que será validado. São utilizadas as funções para fazer a validação da quantidade de caracteres e dos campos obrigatórios. Ainda, são validados os campos de data do seguro e, para isso, foi necessário criar uma função (linha 170) que realiza a validação tanto para confirmar que é uma data válida quanto para garantir que é do ano atual. Para os campos do tipo *select*, foi inserido o atributo *required* para garantir que o usuário selecione um dos valores fornecidos.

Listagem 9 – Validação do seguro

```
170 $.validator.addMethod("dataValid", function (value, element) {
171         var dataT = value.split('/');
172         var hoje = new Date();
173         hoje = hoje.getFullYear();
174         var msgDia = "Data invalida!"
175         if (dataT[1] > 0 && dataT[1] < 13 && dataT[0] > 0 && dataT[2]
    == hoje) {
176             if (dataT[1] == 04 || dataT[1] == 06 || dataT[1] == 9 ||
    dataT[1] == 11) {
177                 if (dataT[0] < 31) {
178                     return true;
179                 } else {
180                     msgDia;
181                     return false;
182                 }
183             } else if (dataT[1] == 2) {
184                 if (dataT[0] < 30) {
185                     return true;
186                 } else {
187                     msgDia;
188                     return false;
189                 }
190             } else {
191                 if (dataT[0] < 32 && dataT[1] != 2) {
192                     return true;
193                 } else {
194                     msgDia;
195                     return false;
196                 }
197             }
198         } else {
199             msgDia;
200             return false;
201         }
202         return true
203     }, msgDia);
204
```

```
205     $(function () {
206         $('#formNewEdit').validate({
207             rules: {
208                 valor: {
209                     maxlength: 45,
210                     minlength: 4,
211                     required: true
212                 },
213                 dataSeg: {
214                     dataValid: true,
215                     required: true
216                 }
217             },
218             submitHandler: function (form) {
219                 form.submit();
220             }
221         });
222     })
```

Fonte: Autoria própria

A Listagem 10 exemplifica como o *controller* da página de relatório de seguros por cliente foi desenvolvido. O modelo dos relatórios foi desenvolvido com o JasperReport, que facilita a elaboração de relatórios sendo necessário apenas informar as tabelas do banco de dados que serão utilizadas e realizar a consulta apropriada para inserir os campos necessários. No *controller* foram desenvolvidos dois métodos correspondentes ao relatório. Na linha 49 da Listagem 10 está o método responsável por apresentar o relatório na página e por efetuar o *download* e, na linha 61, está o método para enviar o relatório por *e-mail*.

Listagem 10 - RelatorioClienteController

```
1   package utfpr.edu.br.tcc.controller;
2
3   import ...
32
33  @Controller
34  @RequestMapping("/relatorio/cliente")
35  public class RelatorioClienteController {
36
37      @Autowired
38      private SeguroRepository repository;
39      @Autowired
40      private SeguroReportService seguroReportService;
41      @Autowired
42      private GerarRelatorio gerarRelatorio;
43      @Autowired
44      private EnvioEmail envioEmail;
45
46      ModelAndView modelAndView = new
  ModelAndView("relatorios/relatorioCliente");
47
48      @GetMapping("/seguro")
49      public void export(@RequestParam("cliente") String cliente,
  @RequestParam(value = "botao") String botao,
50                         HttpServletResponse response,
```

```
  RedirectAttributes attributes)
51              throws IOException, JRException, SQLException {
52          JasperPrint jasperPrint = seguroReportService
  .generatePromissoria(1L, "Relatório de Seguros por Cliente", cliente,
  "classpath:/reports/ClienteSeguroReport.jrxml", "", "");
53          if (botao.equalsIgnoreCase("mostrar")) {
54              gerarRelatorio.imprimir(response, jasperPrint);
55          } else if (botao.equalsIgnoreCase("baixar")) {
56              gerarRelatorio.baixar("RelatorioCliente.pdf", response,
  jasperPrint);
57          }
58          attributes.addFlashAttribute("mensagem","Relatorio gerado com
  sucesso!");
59      }
60      @GetMapping("/email")
61      public void email(@RequestParam("cliente") String cliente,
  @RequestParam("endereco") String endereco,
62                        @RequestParam("assunto") String assunto,
  @RequestParam("texto") String texto,
63                        HttpServletResponse response) throws
  IOException, JRException, SQLException {
64          JasperPrint jasperPrint = seguroReportService
  .generatePromissoria(1L, "Relatório de Seguros por Cliente", cliente,
  "classpath:/reports/ClienteSeguroReport.jrxml", "", "");
65          gerarRelatorio.imprimir(response, jasperPrint);
66          envioEmail.enviarArquivo(endereco, assunto, texto,
  gerarRelatorio.gerarPdf(jasperPrint), "RelatorioCliente.pdf");
67      }
68  }
```

Fonte: Autoria própria

Para que ambos os métodos possam gerar os relatórios é chamado um método em comum nas linhas 52 e 64, no qual são passados os valores necessários como, o título e subtítulo, o elemento a ser filtrado no relatório (nesse exemplo sendo o nome do cliente) e o caminho do arquivo criado pelo JasperReport. Esse método foi desenvolvido na classe *SeguroReport* apresentada na Listagem 11, que se encarrega de pegar os parâmetros passados pelos *controllers* de relatórios e com a assistência do JasperPrint gerar o relatório necessário.

Listagem 11 - SeguroReport

```
1   package utfpr.edu.br.tcc.report.decorator;
2
3   import ...
24
25  @Transactional
26  @Repository
27  public class SeguroReport {
28
29      @Autowired
30      @Qualifier("jdbcTemplate")
31      private JdbcTemplate jdbcTemplate;
32      @Autowired
33      private ResourceLoader resourceLoader;
34
35      public JasperPrint generatePromissoria(Long id, String titulo,
```

```
  String descricao, String caminho, String ordem, String sub)
36      throws SQLException, JRException, IOException {
37
38          Connection conn = jdbcTemplate
  .getDataSource().getConnection();
39          String path = resourceLoader
  .getResource(caminho).getURI().getPath();
40          JasperReport jasperReport = JasperCompileManager
  .compileReport(path);
41          Map<String, Object> parameters = new HashMap<>();
42          parameters.put("TITULO", titulo);
43          parameters.put("ID", id);
44          parameters.put("DESCRICAO", descricao);
45          parameters.put("ORDEM", ordem);
46          parameters.put("SUBTITULO", sub);
47          JasperPrint print = JasperFillManager.fillReport(jasperReport,
  parameters, conn);
48          return print;
49      }
66  }
```

Fonte: Autoria própria

Para que o primeiro método da Listagem 10 possa mostrar e fazer o *download* do relatório são utilizados dois métodos que se encontram na classe GerarRelatorio apresentada na Listagem 12. Para disponibilizar o relatório na tela do usuário é utilizado o método imprimir na linha 31, que recebe o arquivo e o converte em PDF e, por meio do JasperExportManager é exportado o arquivo. Na página HTML dos relatórios é utilizado o componente *iframe* para apresentar o PDF do relatório sem precisar ser redirecionado.

Para fazer o *download* do relatório é utilizado o método baixar da linha 38, que além de receber o arquivo, também recebe o nome que será definido com o formato e com o auxílio do JasperExportManager é exportado o arquivo.

Listagem 12 - GerarRelatorio

```
1   package utfpr.edu.br.tcc.service;
2
3   import ...
27
28  @Service
29  public class GerarRelatorio {
30
31      public void imprimir(HttpServletResponse response,
  JasperPrint jasperPrint)
32              throws IOException, JRException{
33          response.setContentType("application/pdf");
34          OutputStream out = response.getOutputStream();
35          JasperExportManager.exportReportToPdfStream(jasperPrint, out);
36      }
37
38      public void baixar(String nomeArquivo, HttpServletResponse
  response, JasperPrint jasperPrint)
39              throws IOException, JRException {
40          response.setContentType("application/x-download");
```

```
41          response.setHeader("Content-Disposition",
42          String.format("attachment; filename=\""+nomeArquivo+"\""));
43          OutputStream out = response.getOutputStream();
44          JasperExportManager.exportReportToPdfStream(jasperPrint, out);
45      }
46  }
```

Fonte: Autoria própria

Para enviar o relatório por *e-mail* foi desenvolvida a classe EnvioEmail exposta na Listagem 13, que possui dois métodos para envio de *e-mail*, um para enviar apenas mensagens e outro com a possibilidade de anexar um arquivo ao *e-mail*.

Em ambos os casos é utilizado o JavaMailSender para compor o *e-mail*, passando o endereço a ser enviado, o assunto e a mensagem do *e-mail*. Para enviar o relatório pelo *e-mail* é utilizado o segundo método, no qual é empregado o MimeMessage que permite anexar arquivos ao *e-mail*, necessitando apenas do arquivo e o nome.

Listagem 13 - EnvioEmail

```
1   package utfpr.edu.br.tcc.service;
2
3   import ...;
12
13  @Service
14  public class EnvioEmail {
15
16      @Autowired
17      private JavaMailSender mailSender;
18
19      public void enviarMensagem(String endereco, String assunto,
  String menssagem){
20          SimpleMailMessage email = new SimpleMailMessage();
21          email.setTo(endereco);
22          email.setSubject(assunto);
23          email.setText(menssagem);
24          mailSender.send(email);
25      }
26
27      public void enviarArquivo(String endereco, String assunto,
  String menssagem, byte[] bytes, String nomeArquivo){
28          MimeMessage message = mailSender.createMimeMessage();
29          try {
30              MimeMessageHelper helper = new MimeMessageHelper(message,
  true);
31              helper.setTo(endereco);
32              helper.setSubject(assunto);
33              helper.setText(menssagem);
34              helper.addAttachment(nomeArquivo, new
  ByteArrayResource(bytes));
35              mailSender.send(message);
36          } catch (MessagingException e) {
37              e.printStackTrace();
38          }
39      }
40  }
```

Fonte: Autoria própria

5 CONCLUSÃO

O objetivo principal deste trabalho foi desenvolver um sistema *web* para auxiliar os corretores no controle e gerenciamento dos pagamentos de seguro. Para isso foram analisados os requisitos e as necessidades referentes às funcionalidades que o sistema deveria dispor.

Existem várias corretoras de seguro franquiadas, que possuem um sistema para a efetivação de seguros fornecidos pela própria empresa, mas não são todas que possuem um sistema para o controle do pagamento, tanto para os seguros quanto para suas parcelas. Nesses casos, fica em função dos funcionários criarem planilhas e tabelas para manter esses registros atualizados.

Assim, esse sistema teve como objetivo atender a esta necessidade possibilitando cadastrar e acompanhar o andamento dos seguros e permitindo que o corretor tenha um meio simples e ágil de alertar o cliente a respeito de seus pagamentos. Além de possibilitar que o próprio cliente possa ter acesso ao sistema para visualizar e acompanhar os pagamentos de seus seguros.

Diversas ferramentas foram utilizadas no desenvolvimento desse projeto, como o Bootstrap, que é um *framework* para auxiliar a elaboração das páginas *web*, fornecendo componentes simples, fáceis de serem utilizados e eficientes em relação ao *design* e funcionalidades desejadas. O Thymeleaf, que facilitou o desenvolvimento por oferecer a possibilidade de reutilização de código, evitando que precise escrever em todas as páginas o mesmo código, já que é possível associar fragmentos de diversas páginas para formar um leiaute padronizado. O Spring, que fornece várias ferramentas tanto para o auxílio no desenvolvimento do projeto quanto para garantir a segurança dos dados cadastrados, pois são realizados cadastro dos clientes e suas informações devem permanecer armazenadas de forma segura.

Após o desenvolvimento do sistema, verificou-se que os objetivos definidos para este projeto foram alcançados, atendendo as necessidades do corretor e do cliente, fornecendo uma maneira mais fácil, ágil e segura de controlar os pagamentos de cada cliente e permitindo que o cliente possa se manter informado por meio do acesso à página dos dados dos seus pagamentos.

REFERÊNCIAS

BERNERS-LEE, T. et al. The World-Wide Web. Communications of the ACM, New York, v.37, n.8, p.76-82, Aug. 1994.

cgi.br. **TIC Empresas 2012 revela que metade das empresas de grande porte já usam as redes sociais**. 2013. Disponível em: < http://www.cgi.br/noticia/tic-empresas-2012-revela-que-metade-das-empresas-de-grande-porte-ja-usam-as-redes-sociais/>. Acesso em: 28 mar. 2018.

DOOLEY, John. **Software development and professional practice**. 1ª edição, Apress, 2011

FERRANTE, A. J.; RODRIGUEZ, M. V. R. **Tecnologia de Informação e gestão empresarial**. 2ª ed., E-Papers: Rio de Janeiro, 2004.

MARTINS, Jose Carlos Cordeiro. **Técnicas para gerenciamento de projetos de software**. 1ª edição, BrasPort: Rio de Janeiro, 2007.

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

RAMAKRISHNAN, R.; GEHRKE, J. **Sistemas de gerenciamento de banco de dados**. 3ª edição AMGH Editora: Porto Alegre, 2008.

SAMPAIO, Cleuton. **Web 2.0 e Mashups:** Reinventando a Internet. 1ª edição, BrasPort: Rio de Janeiro, 2007.

SCHUNCKE, Alex. **Comunicação de dados síncrona x assíncrona**. 2013. Disponivel em:<https://www.oficinadanet.com.br/post/9978-comunicacao-de-dados-sincrona-x-assincona>. Acesso em: 11 de maio 2018.

SEBRAE. **Corretora de seguros**. 2018. Disponível em: http://www.sebrae.com.br/sites/PortalSebrae/ideias/como-montar-uma-corretora-de-seguros,f7887a51b9105410VgnVCM1000003b74010aRCRD. Acesso em: 20 mar. 2018.

SILVEIRA, P. et al. **Introdução à arquitetura de design de software:** Uma Visão Sobre a Plataforma Java. 1ª edição, Elsevier Brasil: São Paulo, 2011.

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