FACULDADE DE TECNOLOGIA DE AMERICANA
Curso Superior de Análise e Desenvolvimento de Sistemas

Alexandre Ribeiro Makiyama

SISTEMA DE GERENCIAMENTO DE CHANGELOG

Americana, SP
2018

FACULDADE DE TECNOLOGIA DE AMERICANA
Curso Superior de Análise e Desenvolvimento de Sistemas

Alexandre Ribeiro Makiyama

SISTEMA DE GERENCIAMENTO DE CHANGELOG

Trabalho de Conclusão de Curso desenvolvido em cumprimento à exigência curricular do Curso Análise e Desenvolvimento de Sistemas, sob a orientação do (a) Prof. Esp. Antônio Alfredo Lacerda Área de concentração: Desenvolvimento de sistemas.

Americana, SP
2018

FICHA CATALOGRÁFICA – Biblioteca Fatec Americana - CEETEPS
Dados Internacionais de Catalogação-na-fonte

M195s MAKIYAMA, Alexandre Ribeiro

Sistema de Gerenciamento de Changelog. / Alexandre Ribeiro Makiyama. – Americana, 2018.

55f.

Monografia (Curso de Tecnologia em Análise e Desenvolvimento de Sistemas) - - Faculdade de Tecnologia de Americana – Centro Estadual de Educação Tecnológica Paula Souza

Orientador: Prof. Esp. Antônio Alfredo Lacerda

1 Desenvolvimento de software 2. Java – linguagem de programação I. LACERDA, Antônio Alfredo II. Centro Estadual de Educação Tecnológica Paula Souza – Faculdade de Tecnologia de Americana

CDU: 681.3.05
681.3.061

Alexandre Ribeiro Makiyama

SISTEMA DE GERENCIAMENTO DE CHANGELOG

Trabalho de Conclusão de Curso desenvolvido em cumprimento à exigência curricular do Curso Análise e Desenvolvimento de Sistemas, sob a orientação do (a) Prof. Esp. Antônio Alfredo Lacerda Área de concentração: Desenvolvimento de sistemas.

Americana, 04 de Dezembro de 2018.

Banca Examinadora:

Antônio Alfredo Lacerda
Especialista
Fatec Americana

André de Lima
Doutor
Fatec Americana

Alberto Martins Júnior
Mestre
Fatec Americana

RESUMO

Este trabalho apresenta o conceito do que é um *changelog* (registro de mudanças feitas em um *software*), expondo sua importância para os usuários e desenvolvedores de sistema. Também desenvolve um sistema *web* para gerenciar e distribuir os *changelogs* de *software*, com objetivo de facilitar a manutenção e organização de entradas ou mudanças do mesmo, possibilitando também a publicação do changelog com opções de permissionamento para o que os usuários possam avaliar e acompanhar as entradas, assim facilitando a distribuição de informação relevante para os usuários. O sistema deste trabalho foi desenvolvido utilizando o gerador de aplicações *JHipster*, que fornece ferramentas para o desenvolvimento de sistemas, ferramentas como o *Spring Boot* que é um *Framework* para plataformas Java no servidor e o *Angular* com *Bootstrap* no cliente. Ambas as ferramentas citadas são utilizadas no desenvolvimento do sistema apresentado por este trabalho. Para o banco de dados o *JHipster* oferece diversas opções como *PostgreSQL*, *Cassandra*, *MongoDB*, *MySQL*, entre outras. O Banco de dados escolhido para este trabalho foi o *MongoDB*, que é um banco de dados não relacional orientado a documentos JSON para armazenar os dados. Explicando um pouco sobre cada ferramenta utilizada a fim de se propagar o conhecimento sobre tais recursos.

Palavras-chave: Desenvolvimento *web*, *Changelog*, *JHipster*.

ABSTRACT

This paper conceptualizes changelogs (history of changes made to a software), explaining its importance to systems users and developers. It also develops a web application to manage and publish a software changelog, with the objective of facilitating the maintenance and organization of the changelog entries, also enabling the changelog publication with level of permissions so users may evaluate and follow entries, facilitating the distribution of relevant information to users. This system was developed using an application generator Jhipster, which provides resources for software development, resources like Sping Boot which is a framework for Java back-end development and Angular with Bootstrap which are for front-end development or client. Both these frameworks are used in this paper to develop the system proposed by it. For its database the Jhipster framework provides an array of options like PostgreSQL, Cassandra, MongoDB, MySQL and others. The database chosen for this system was MongoDB which is a no-sql database, oriented to documents JSON to store data. Explaining a little about each framework to spread the knowledge about such resources.

Keywords: Web development, *Changelog, JHipster*.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

BSON	*Binary JSON*
CRUD	*Create, read, update e delete*
HTML	*Hypertext Markup Language*
JDL	*Jhipter Domain Language*
JSON	*JavaScript Object Notation*
MIT	*Massachusetts Institute of Technology*
MVC	*Model-View-Controller*
SPA	*Single Page Application*
SQL	*Structured Query Language*
UML	*Unified Modeling Language*

INTRODUÇÃO

Este trabalho irá detalhar os *changelogs* de *software* e sua importância para o desenvolvimento de aplicações e sua utilização na rastreabilidade de mudanças efetuadas em sistemas. O trabalho também abordará o desenvolvido uma aplicação web para o gerenciamento de *changelogs*.

O objetivo é explicar o que é um *changelog* e demonstrar como utiliza-lo pode beneficiar o processo de desenvolvimento, manutenção e comercialização de um sistema e desenvolver uma aplicação para gerenciar e disponibilizar *changelog* de sistemas.

Para descrever os changelogs este trabalho faz uso de duas referências principais, primeiro o GNU *Coding Standards* (Padrões de programação GNU) escrito por Richard Stallman e contribuidores pela internet. A segunda referência é o projeto *Keep a Changelog* (Mantenha um changelog) que pode ser considerado uma atualização sobre alguns pontos do *Coding Standards* sobre changelogs.

No desenvolvimento do gerenciador de changelogs será utilizado o *Framework* de desenvolvimento *JHipster*, que foi utilizado para a criação do modelo inicial da aplicação, *back-end* e *front-end*. A aplicação será estruturada no modelo MVC que será descrito no capítulo que fala sobre o desenvolvimento da aplicação.

Para clarificar este trabalho enfatiza dois pontos: descrição do que são *changelogs*, porque usá-los e desenvolvimento de um sistema para gerenciar esses changelogs.

A ideia deste trabalho foi concebida através da experiência que os desenvolvedores, usuários e pessoas envolvidas nos projetos de desenvolvimento de *software* vivenciam ao precisar saber o que modificado no sistema, como por exemplo: saber quais são novos recursos para uma apresentação comercial, tabelas que foram modificadas, recursos removidos ou adicionados. Com o desenvolvimento da aplicação deste trabalho espera-se contribuir com o gerenciamento das mudanças em sistemas, fazendo com que essas informações estejam disponíveis quando necessárias.

No primeiro Capítulo é descrito o que são os *changelogs* de sistemas, a origem da ideia, é feita uma analogia com o gerenciamento de configuração descrito por Sommerville e são dados alguns exemplos de *changelogs* em projetos disponíveis na internet, no segundo Capítulo apresenta uma descrição da engenharia de *software*, quais conceitos desta foram utilizados neste trabalho, como engenharia de requisitos e diagramas utilizados para representar os objetos e funcionalidades do sistema, no terceiro Capítulo são apresentados os requisitos de usuário e de sistema que foram levantados para o desenvolvimento do sistema proposto, utilizando a especificação destes requisitos são apresentados os diagramas UML de classe, atividade e sequência e quarto Capítulo: apresenta mais detalhadamente os *Frameworks* utilizados no desenvolvimento do sistema, demonstra como a aplicação é gerada, detalha a estrutura do sistema e prossegue descrevendo o processo de desenvolvimento.

Metodologia

A metodologia de pesquisa utilizada neste trabalho é exploratória, visando reunir dados sobre o assunto utilizando documentações, projetos como os já mencionados *GNU Coding Standards* e o *Keep a Changelog*, e analisando grupos de discussão *online* sobre o assunto *changelogs*, para que ao final desta pesquisa seja desenvolvido um sistema que contribua para a solução dos problemas encontrados.

1. CHANGELOG

Este capítulo tratará de descrever o que é o *changelog*, sua função no desenvolvimento e manutenção de *software*, a sua estrutura e organização, como é utilizado e sua importância geral. Para isso serão utilizadas como referências o GNU *Coding Standards*, o projeto *Keep a Changelog* e o livro Engenharia de *Software* escrito por Ian Sommerville (2013).

Changelog é um documento que contém registros das mudanças desenvolvidas em um sistema. O GNU *Coding Standards* na seção de documentação diz que se pode pensar no changelog como uma lista de tarefas realizadas, nas quais se explica como as versões anteriores diferem da atual, também diz que apesar dos usuários verem o sistema como está na versão mais atual, os mesmos querem uma explicação clara do que foi modificado.

Tais registros auxiliam desenvolvedores quando, por exemplo, vão programar uma funcionalidade e precisam analisar seu impacto no projeto, ou quando precisam tratar falhas no sistema. Isso porque como se pode ver no livro Engenharia de *Software*: "Os sistemas de *software* sempre mudam durante seu desenvolvimento e uso. *Bugs* são descobertos e precisam ser corrigidos. Os requisitos do sistema mudam e é preciso implementar essas mudanças em uma nova versão do sistema" (SOMMERVILLE, 2013, Engenharia de *Software*, p. 475).

Devido a essa característica dos sistemas de *software*, viu-se que é necessário, manter o registro das mudanças realizadas, para servir como referência no desenvolvimento e melhorar o gerenciamento de versões. Isto porque como também diz Sommerville é fácil para as pessoas esquecerem o que foi trabalhado, também adicionando o fato de que hoje muitas equipes não trabalham mais no mesmo local, podendo estar em locais geograficamente distantes uns dos outros, tornando às vezes difícil a comunicação entre os membros do projeto.

No livro descreve-se uma nova versão como: "Uma instância de um item de configuração que difere, de alguma forma, de outras instâncias deste item. As versões sempre têm um identificador único, o qual é geralmente composto pelo nome do item de configuração mais um número de versão." (SOMMERVILLE, 2013, p. 477). Ou seja, uma modificação realizada no sistema, identificada pelo nome do recurso e o número de versão do mesmo.

Pode-se então dizer que a versão descrita por Sommerville é análoga a um registro de *changelog* no qual também se encontra uma descrição da mudança.

Para completar a analogia que está sendo realizada entre *changelog* e uma parte do gerenciamento de configuração descrito por Sommerville, ainda é preciso dar a descrição de *release*, que em engenharia de *software* se encontra como: "Uma versão de um sistema que foi liberada para os clientes (ou outros usuários em uma organização) para uso." (SOMMERVILLE, 2013, Engenharia de *Software*, p. 477). Como o *changelog* proposto no GNU - *Coding Standards* e os demais já presentes em projetos só tratam das mudanças já implementadas no sistema, pode-se dizer então que é equivalente à um *release*. Com isso pode-se dizer então que o *release* descrito pela engenharia de *software* é o *changelog*, e os registros são as entradas de um *changelog*.

Realizada então essa analogia e com a descrição dada, pode-se definir *changelog* como o histórico das mudanças desenvolvidas sobre um sistema com uma explicação do motivo da mudança ter ocorrido e quando necessário uma descrição de como a mudança irá funcionar, alguns exemplos:

- Método (ou função) x teve sua chamada alterada, porque um de seus argumentos não era utilizado, agora sua chamada passa a ser com k argumentos.
- A funcionalidade y foi removida do sistema, pois se tornou obsoleta.
- Coluna w foi adicionada na tabela z, nesta coluna serão armazenados descritivos.

Para esses exemplos dá se a denominação de entrada, que também podem ser agrupadas resultando em um *change set* (termo encontrado no GNU *Coding Standards*), ou conjunto de mudanças.

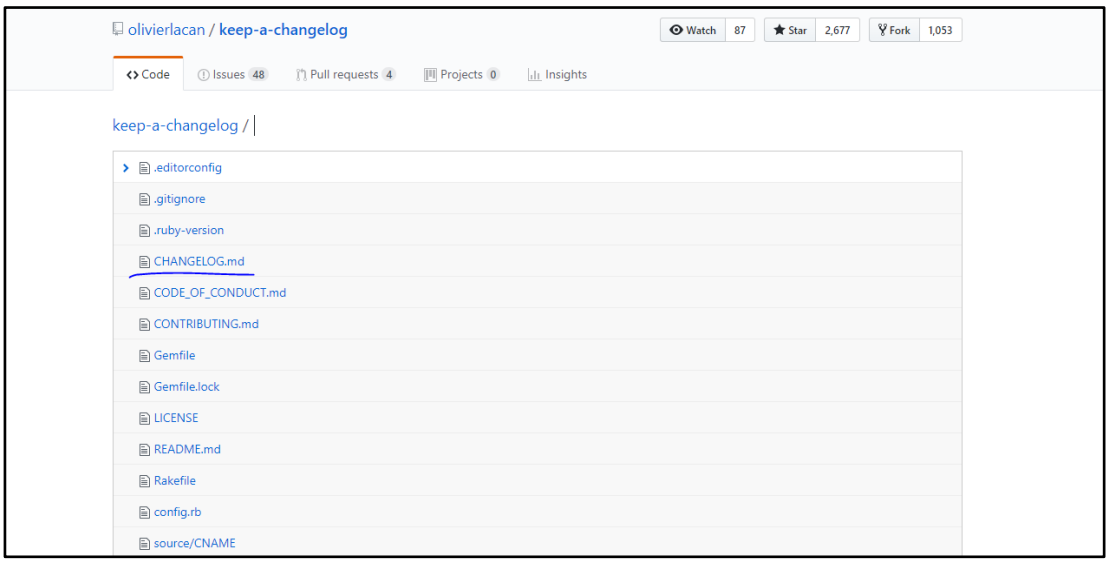
Figura 1 - Richard Stallman coloca como exemplo de *changelog*.

```
1998-08-17  Richard Stallman  <rms@gnu.org>

* register.el (insert-register): Return nil.
(jump-to-register): Likewise.

* sort.el (sort-subr): Return nil.

* tex-mode.el (tex-bibtex-file, tex-file, tex-region):
Restart the tex shell if process is gone or stopped.
(tex-shell-running): New function.

* expr.c (store_one_arg): Round size up for move_block_to_reg.
(expand_call): Round up when emitting USE insns.
* stmt.c (assign_parms): Round size up for move_block_from_reg.
```

Fonte: GNU Coding Standards, section 6.8.2 Style of Change Logs.

Na figura 1, pode-se observar a data das mudanças, seu autor e as descrições das alterações. O exemplo mostrado foi feito através do *Emacs*, editor de texto desenvolvido para o sistema operacional GNU. Geralmente *changelogs* são simplesmente isso, um arquivo .txt ou .md dentro do projeto nos quais se registram as mudanças realizadas pelos programadores. Para exemplificar essa situação a figura 2 mostra a estrutura de arquivos do projeto *Keep changelog*.

Figura 2 - Exemplo de arquivo changelog.md do projeto *Keep a Changelog*.



Fonte: https://github.com/olivierlacan/keep-a-changelog/find/master, Acesso em: setembro de 2018.

Como o projeto do exemplo tem uma licença MIT e está disponível no *GitHub*, que é uma plataforma de versionamento de código fonte na nuvem. Neste caso o arquivo está acessível para qualquer pessoa que se busque saber quais foram as modificações realizadas, porém nem todo sistema dispõe, quer ou pode dispor dessa visibilidade para qualquer indivíduo, então o *changelog* permanece apenas como arquivo referência de mudanças para

desenvolvedores. E como fazem os demais usuários ou pessoas interessadas para saber o que foi modificado? Em muitos casos estes ficam dependentes de um desenvolvedor para verificar o arquivo *changelog* do sistema ou recorrer a sua memória e lembrar o que foi modificado.

Em alguns casos o sistema pode não possuir um *changelog* ou este não ficar disponível aos usuários. O aplicativo *Spotify* é um exemplo. Em seu fórum pode-se encontrar uma discussão aberta por um de seus usuários, disponível em https://community.spotify.com/t5/Desktop-Windows/Release-Notes-Changelog/td-p/4405324. Alguns usuários reclamam sobre a falta de informação que o aplicativo fornece quanto a suas alterações, eles apontam para principalmente 2 pontos, primeiro: com a falta da informação perdem a segurança de como está sendo gasto o tempo da equipe de desenvolvimento com questões importantes, e segundo: existem usuários que gostariam de saber o que foi implementado afim de melhor entender as alterações que foram feitas e se familiarizar com o produto. Porém a cada atualização realizada é sempre publicada uma mesma mensagem padrão.

Outra publicação trata de explicar o que são *changelogs* e como cada empresa manuseia o seu, até mesmo apontando como problema dos *changelogs:* "O problema com esses changelogs vêm de serem normalmente difíceis de ler e entender a menos que esteja diretamente envolvido com o projeto, ou seja, um desenvolvedor treinado em ler jargões e documentação". (KEETON, *Changelog and Release Notes Explained for Non-Developers*, 2017, Tradução: Autor). Em reposta a essa publicação alguns usuários criticaram o conteúdo exposto, apontando principalmente a frequência de atualizações feita pela empresa e a dificuldade de se encontrar os *changelogs* no *site*.

Com o tempo aplicações crescem, os envolvidos no projeto mudam e se torna difícil acompanhar as mudanças efetuadas. Para gerenciar essas mudanças, é então proposto o desenvolvimento de um sistema para o gerenciamento de mudanças seguindo o formato de *changelog*, mais especificamente o formato e sugestões descritas no projeto *Keep a changelog*.

2. ENGENHARIA DE *SOFTWARE*

A engenharia de *software* é descrita como: "... disciplina de engenharia cujo foco está em todos os aspectos da produção de *software*, desde os estágios iniciais da especificação até sua manutenção…" (SOMMERVILLE, 2013, Engenharia de *Software*, p. 5). Desta maneira a engenharia de *software* será utilizada neste trabalho para fazer a especificação inicial do sistema e guiar a atividade de desenvolvimento utilizando seus processos.

Sommerville descreve os processos de *software* como sendo o conjunto de atividades que levam a produção de um produto de *software*, indicando quatro atividades fundamentais em qualquer projeto de *software*: sendo elas a especificação, implementação, validação e evolução (SOMMERVILLE, 2013, Engenharia de *Software*, p. 18).

A especificação de *software* é descrita como: "A funcionalidade do *software* e as restrições a seu funcionamento devem ser definidas." (SOMMERVILLE, 2013, Engenharia de *Software*, p. 18). Isto é a especificação inicial do projeto onde geralmente o engenheiro ou analista de sistemas faz a entrevista com o cliente para saber quais necessidades o sistema deve atender, ditando a forma que o sistema vai ser desenvolvido, através das representações (diagramas) que são elaborados utilizando a especificação levantada.

Após a especificação é iniciada a atividade de implementação de *software*, e é nesta atividade que o sistema é desenvolvido. No Capítulo quatro é descrito o desenvolvimento do sistema. A validação compreende a atividade de validar o sistema a fim de ver se o mesmo foi desenvolvido conforme a especificação, a última atividade citada é a evolução, na qual são contempladas novas mudanças no sistema. Essas duas últimas atividades são utilizadas na conclusão deste trabalho para analisar se o sistema atende as especificações e que alterações poderiam ser realizadas.

2.1 ESPECIFICAÇÃO DE REQUISITOS

A especificação dos requisitos de sistema é a descrição das suas funcionalidades, descrevendo como estas irão funcionar para o usuário ou *stakeholder* de sistema, relacionando a elas possíveis restrições quanto a sua utilização e funcionamento, ou como Sommerville coloca em seu livro engenharia de *software*: "Os requisitos de um sistema são as descrições do que o sistema deve fazer os serviços que oferece e as restrições a seu funcionamento." (SOMMERVILLE, 2013, Engenharia de *Software*, p.57).

2.2 DIAGRAMAS

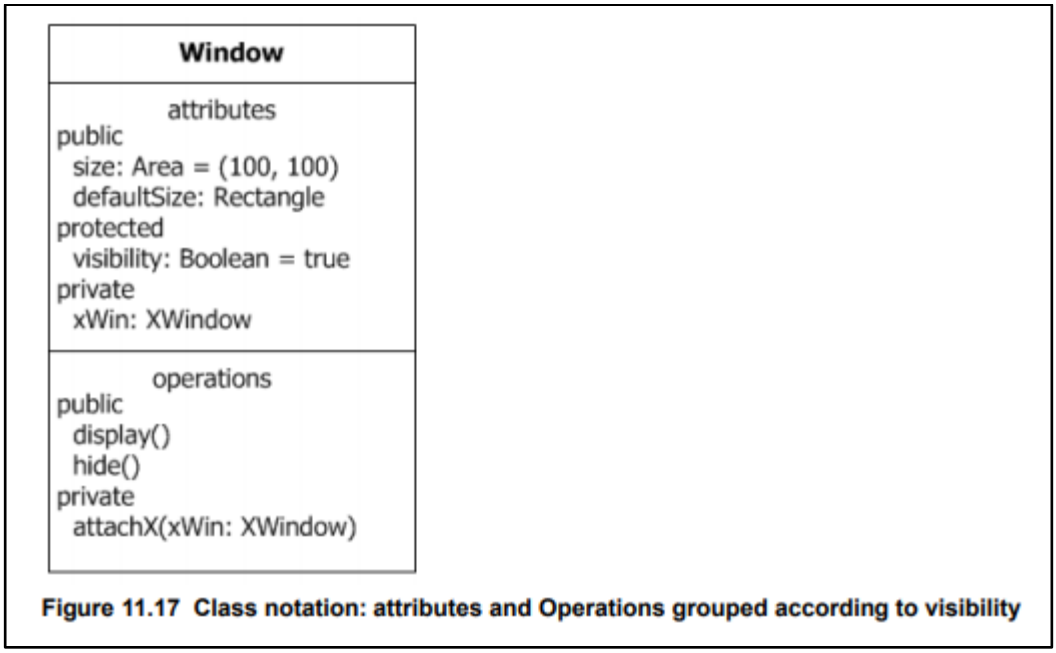
Esta seção trata dos diagramas que irão ser utilizados no processo de desenvolvimento do sistema. No caso são apresentados três diagramas: classe, atividades e sequência, a descrição destes diagramas segue a especificação mais recente da UML a versão 2.5.1. Depois de especificados esses diagramas, os mesmos serão utilizados no Capítulo três para especificar o sistema proposto.

Diagrama de classes

Na especificação da UML o diagrama de classes é descrito como a especificação de uma classificação de objetos, propriedades e comportamento dos mesmos (OMG, OMG *Unified Modeling Language* (OMG UML), pág. 194). Ou seja, a especificação das classes do sistema com seus atributos e métodos. Sendo as propriedades os atributos, por exemplo, um id. E comportamento os métodos da classe, que são as ações que o objeto pode executar.

Na especificação pode-se encontrar a seguinte representação de um diagrama de classes representado pela figura 3.

Figura 3 - Diagrama de classes - Classe.



Fonte: OMG Unified Modeling Language (OMG UML), p. 196.

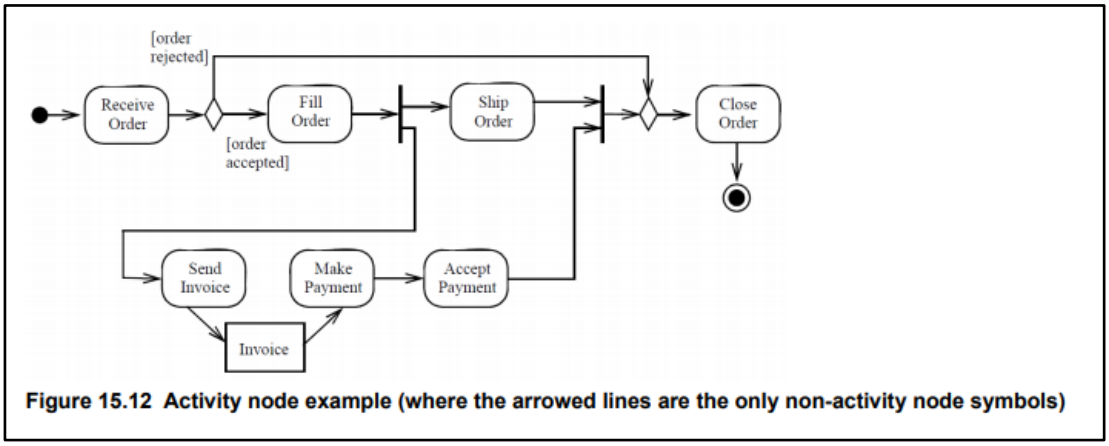
Na figura 3 se pode ver então a especificação de uma classe *Window*, que possui os atributos: *size*, *visibility*, *defaultSize* e *xWin*. Para cada propriedade é atribuído um tipo, nível de acesso (representado por + *public*, - *private* e # *protected*) e um valor padrão. A classe *Window* também possui três operações conhecidas como métodos, *display*, *hide* e *attachX*. Sendo indicado que este último método recebe como parâmetro *xWin* do tipo *XWindow*.

Diagrama de atividades

Seguindo a especificação da UML o diagrama de atividades tem o objetivo de descrever atividades executadas pelo sistema, apresentando como os processos do mesmo serão executados e também para guiar os interessados, pelos processos que o sistema irá executar.

Para o sistema que está sendo desenvolvido neste trabalho este diagrama irá representar os métodos das classes, que são as ações iniciadas pelo usuário e como sistema vai processa-las. No manual da UML é apresentado o seguinte exemplo de um diagrama de atividades.

Figura 4 - Exemplo diagrama de atividades.



Fonte: OMG Unified Modeling Language (OMG UML), p. 196.

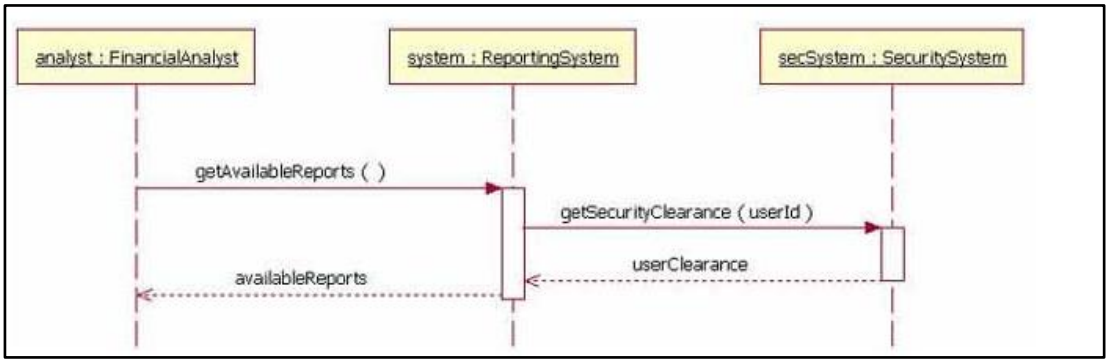
O diagrama apresentado na figura 4 como exemplo descreve a atividade de se fazer um pedido e é iniciado no círculo preto simples ou sólido, passando pelas ações descritas nos retângulos que podem passar por condições representadas pelos triângulos. Uma ação pode se dividir em algum ponto, que é o que acontece na *ship order* (enviar pedido), esse comportamento é ilustrado pela barra preta, mas sempre se unindo novamente em algum ponto depois de enviar pedido ou aceitar pagamento como no diagrama do exemplo. A atividade é finalmente finalizada no encerramento e pelo símbolo do círculo sólido dentro de outro círculo.

Diagrama de sequência

Um diagrama de sequência serve para representar o ciclo de vida dos objetos do sistema dentro de determinada ação executada por um ator. A especificação do UML descreve-os como representação das interações, focando na sequência de mensagens que são trocadas dentro do ciclo de vida do processo (OMG, OMG *Unified Modeling Language* (OMG UML), pág.595).

A figura 5 ilustra um diagrama de sequência, para o processo de visualizar relatórios financeiros.

Figura 5 - Exemplo diagrama de sequência.



Fonte: https://www.ibm.com/developerworks/rational/library/3101.html, Acesso em: setembro 2018.

Este contém o ator *FinancialAnalyst*, analista financeiro e dois objetos do sistema *ReportingSystem* (sistema de relatórios) e *SecuritySystem* (sistema de segurança). O processo se inicia com uma requisição do analista para ver relatórios, o sistema de relatórios então usa o método *getSecurityClearance*() para verificar a permissão do analista, retornando o resultado para o sistema de relatórios que por sua vez retorna os relatórios para o analista.

É claro que existem muitos outros aspectos para serem abordados sobre os diagramas citados, como a especificação dos elementos de associação, ação, condição, agrupamento, etc. Porém o objetivo deste capítulo é apresentar os diagramas que serão utilizados neste trabalho e indicar a referência do trabalho para construir os diagramas do Capítulo seguinte.

3. ESPECIFICAÇÃO DO SISTEMA DE *CHANGELOG*

Utilizando os conceitos apresentados no Capítulo anterior sobre a engenharia de *software*, neste capítulo será apresentado o levantamento dos requisitos para a aplicação que será desenvolvida e os diagramas de classe, atividade e sequência que foram produzidos utilizando a ferramenta *LucidChart* disponível em: https://lucidchart.com.

Na próxima seção já seguem descritos os requisitos funcionais e os de sistema descobertos para o desenvolvimento da aplicação, seguindo a seguinte estrutura:

1. Requisito de usuário: serviço ou funcionalidade do sistema, seguida de sua descrição genérica.
1.1. Requisito de sistema: descrição mais detalhada da funcionalidade e como será implementada.

3.1 REQUISITOS

1. Os usuários poderão se registrar no *website* provendo um endereço de e-mail válido e uma senha. Estes vão ser utilizados no sistema para fazer *login*.
1.1 O sistema vai ter um módulo de usuários, onde na página principal qualquer pessoa registrar uma conta, fornecendo seu endereço de e-mail, *login* e uma senha.

2. Tendo o usuário criado uma conta, ele poderá cadastrar um sistema dele, este sistema inicialmente não deverá ficar visível para os outros usuários, ou seja, assim que for criado o sistema ele ficará com um status privado, no qual apenas o criador poderá ver.
2.1 O sistema vai ter uma tela de cadastro de sistemas, onde o usuário irá digitar as entradas cadastrais como, por exemplo, nome e descrição.
2.2 Cada projeto vai ter um atributo que servirá pra dizer se ele é publicamente visível ou não, e todo projeto assim que criado terá este atributo com o valor *false*, indicando que o sistema não é publicamente visível.

3. Usuários podem se registrar como membros de um sistema cadastrado, deverá haver um gerenciamento de membros dos sistemas cadastrados, assim os *changelogs* podem ser compartilhados entre esses membros de projeto com diferentes níveis de permissionamento.

3.1 Cada usuário deste sistema poderá ser membro do sistema cadastrado, usuários podem encontrar projetos públicos e pedir para se juntar a equipe.

3.2 O sistema terá um gerenciamento de permissões para cada membro do projeto, dizendo se o membro pode, por exemplo: aceitar outros membros, editar páginas de sistemas, publicar changelogs, etc.

3.3 Incrementando ao requisito 2, quando um usuário cadastrar um sistema, automaticamente será gerado um membro com todas as permissões ativadas para ele.

4. Um membro com permissão pode criar um *changelog* para o sistema, fornecendo suas entradas básicas como: título, descrição, status de publicação, etc.

4.1 No sistema será desenvolvido um módulo de gerenciamento de *changelogs*, onde um membro do sistema registrado pode criar/editar/deletar/publicar um *changelog*.

5. A partir de um *changelog* criado um membro com permissão pode cadastrar uma entrada do *changelog*.

5.1 Na tela de *changelogs* usuários com permissão poderão acessar o cadastro de entradas e dependendo da permissão do membro o mesmo vai poder editar/deletar essas entradas.

6. Membros com permissão podem gerenciar os outros membros do sistema cadastrado, podendo mudar suas permissões ou removê-los.

6.1 Criar um módulo de gerenciamento de membros do sistema cadastrado, onde um membro com permissão pode editar as permissões dos demais membros ou removê-los.

6.2 Para que um membro com permissão de edição de permissões não faça essas sobre outro membro que não poderia ter suas permissões editadas, por exemplo: o criador do sistema dá a permissão para um mesmo editar as permissões e esse membro remove o criador do sistema cadastrado. Para cada membro será criado um status dizendo que suas permissões não podem ser editadas ou o mesmo removido do sistema.

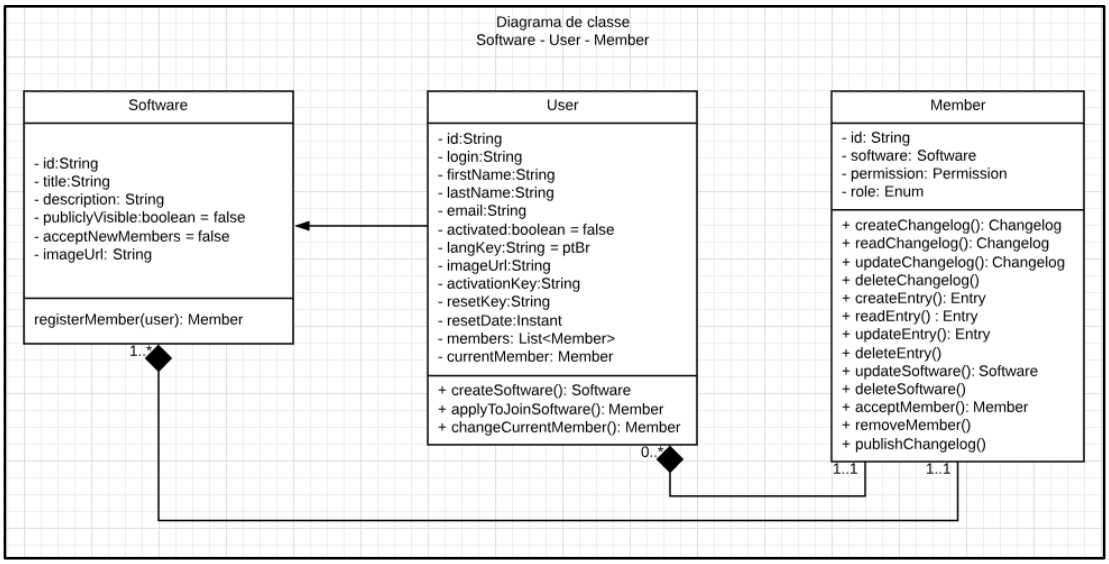
Esses são os seis requisitos que vão servir como base para a construção do projeto, é claro que esses requisitos ainda podem gerar outros, requisitos podem surgir ao decorrer do desenvolvimento da aplicação, ou como já citado: "Os sistemas de *software* sempre mudam durante seu desenvolvimento e uso..." (SOMMERVILLE, 2013, Engenharia de *Software*, p. 475).

3.2 DIAGRAMA DE CLASSES

Então com base nas especificações levantadas na seção dois, é apresentado o diagrama de classes.

Para facilitar o entendimento sobre os objetos que vão constituir o sistema, o diagrama de classes foi quebrado por módulos, que seguem os requisitos. Os diagramas foram criados em inglês, pois ao se desenvolver o sistema é utilizado o inglês para dar nome às classes, atributos e métodos do mesmo. Na figura 6 se encontra o primeiro diagrama com três classes: *User* (usuário) é a classe que representa qualquer usuário cadastrado no sistema. *Member* (membro) representa um membro de um *software* criado por um usuário e *Software* que é a classe representando um programa cadastrado e terá seus *changelogs* gerenciados.

Figura 6 - Diagrama de classe das entidades *User, Member* e *Software*.



Fonte: Autor.

User

Representa um usuário, qualquer pessoa pode se cadastrar no sistema como um *User*, fornecendo um *email*, *login* e senha.

Tabela 1 - Atributos da classe *User*.

Atributo	Descrição
id	Identificador único no sistema.
login	Identificador do usuário para *login* no sistema.
firstName	Primeiro nome do usuário.
lastName	Sobrenome do usuário.
email	E-mail que o usuário usa para se cadastrar, um e-mail só pode ter um usuário associado a ele.
activated	Atributo que serve para indicar se esse usuário foi ativado.
langKey	Idioma em que o sistema aparece para o usuário, inicialmente o sistema vai ter duas línguas, português do Brasil e inglês.
imageUrl	*Url* da imagem que o usuário usa como "foto de perfil".
activationKey	Chave de ativação do usuário, quando um usuário se cadastra um e-mail é enviado para ele com essa chave, com ela o usuário ativa sua conta.
resetKey	Chave de reset de senha serve para recuperar uma conta, caso o usuário

	tenha esquecido sua senha por exemplo.
resetDate	Data do último *reset*.

Fonte: Autor.

Métodos:

createSoftware: a partir de uma tela de cadastro de sistema o usuário cria um *software* para gerenciar.

***applyToJoinSoftware*:** a partir de qualquer projeto que aceite membros novos um usuário pode pedir para ser membro, quando um usuário faz isso um novo membro de projeto é criado e fica aguardando aprovação.

Software

Classe que representa um sistema para ter seus *changelogs* gerenciados, qualquer usuário pode criar um sistema. Atributos:

Tabela 2 - Atributos da classe *Software*.

Atributo	Descrição
id	Identificador único.
title	Um nome dado ao sistema, como por exemplo.
description	Descrição do sistema.
publiclyVisible	Serve para indicar se o projeto deve estar visível para usuários que já não são membros.
acceptNewMembers	Indica se o sistema cadastrado aceita novos membros, usuários só podem se candidatar a serem membros se o valor deste atributo for *true*.

Fonte: Autor.

Método:

***registerMember*:** assim que um usuário registra um sistema para ser gerenciado a classe cria um membro para o mesmo.

Member

Representa os membros de um sistema cadastrado, dentro de um sistema são objetos dessa classe que interagem com os *changelogs*, entradas e outros membros, pode-se pensar neles como avatares dos usuários dentro do sistema cadastrado.

Tabela 3 - Atributos da classe *Member*.

Atributo	Descrição
id	Identificador único de um objeto dessa classe no sistema.
software	*Software* à que este membro faz parte.
role	*Enum* para indicar se o usuário pode ter suas permissões gerenciadas
permission	Associação com um objeto da classe *permission* que serve para armazenar/gerenciar as permissões de um usuário.

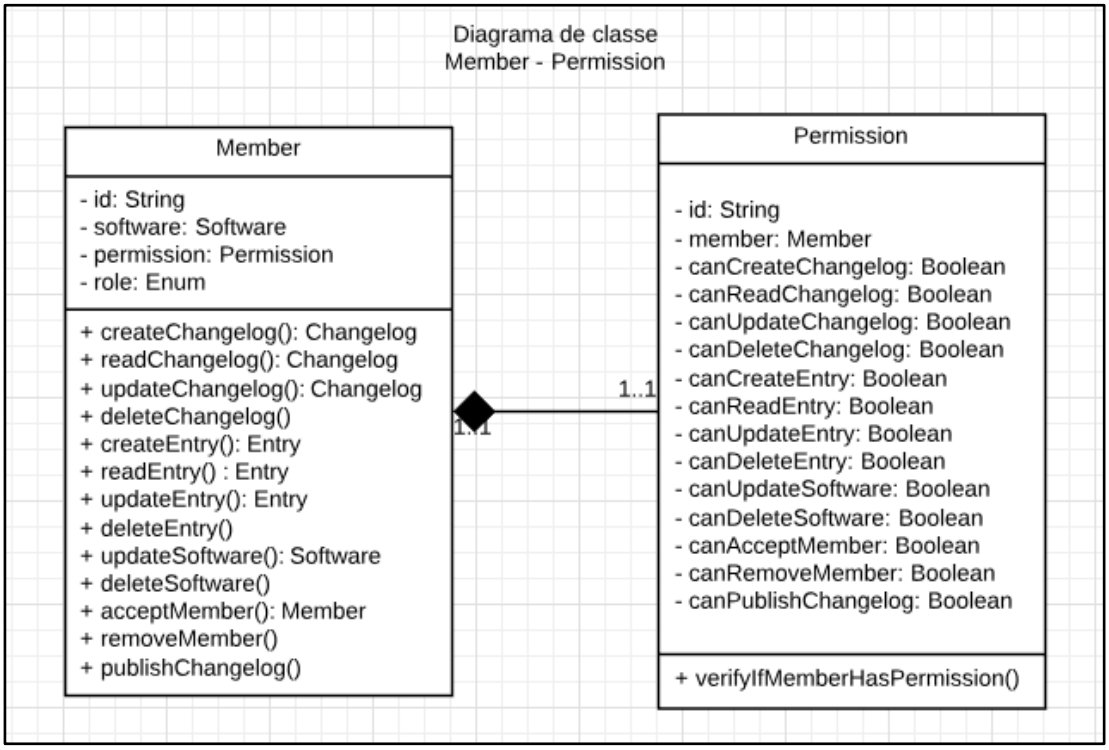
Fonte: Autor.

Métodos:

Os métodos da classe de membros são os que interagem com as outras dentro de um *software*, então para evitar repetição neste trabalho: membros têm todos os métodos que fazem o *CRUD* (*create, read, update* e *delete*) das outras classes. Eles podem também gerenciar as permissões de outros membros e publicar changelogs mediante a permissão.

Para ilustrar o relacionamento do membro com sua permissão, é apresentado na figura 7. Onde se pode ver que cada usuário tem uma permissão e uma permissão é atribuída a um usuário em um relacionamento um para um. Um objeto de permissão tem atributos booleanos que indicam se um membro tem permissão para executar determinada ação, como por exemplo: o atributo *canCreateChangelog*, que é utilizado para saber se o membro pode criar um *changelog*. Para tal a classe possui o método *verifyIfMemberHasPermission*() que verifica a permissão do membro ao tentar executar determinada ação, isto é para cada método *CRUD* no membro (como: *readChangelog, deleteChangelog, publishChangelog*, etc) é verificada a permissão equivalente na classe de permissões.

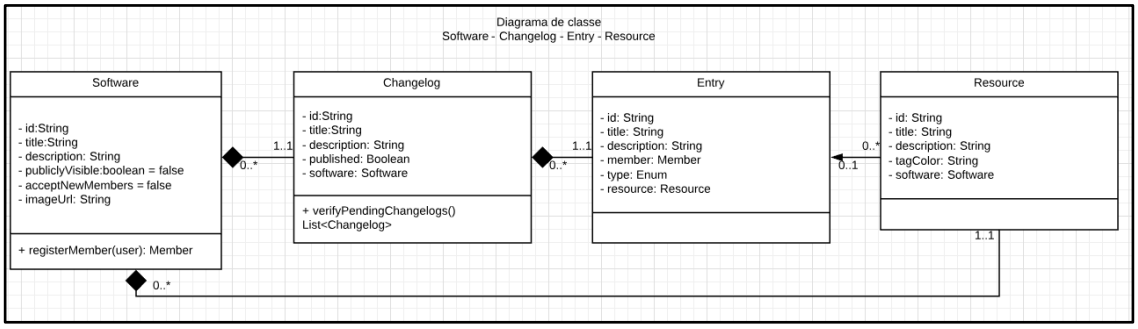
Figura 7 - Diagrama de classe das entidades *Member* e *Permission*.



Fonte: Autor.

No próximo diagrama da figura 8 é apresentada a relação das classes de *Software*, *Changelog*, *Entry* (entrada) e *Resource* (recurso). Cada classe será explicada individualmente, exceto a de *Software* que já foi descrita:

Figura 8 - Diagrama de classe das entidades *Software, Changelog, Entry e Resource*.



Fonte: Autor.

Changelog

É a classe que representa o *changelog*, neste cenário pode ser descrita como grupo de mudanças desenvolvidas no *software*, servindo para agrupar as entradas ou colocando de outra forma, as mudanças realizadas. Com isso são definidos os atributos:

Tabela 4 - Atributos da classe *changelog*.

Atributo	Descrição
id	Identificador de uma instância da classe.
title	Título dado ao changelog, como por exemplo: *changelog* de setembro.
published	Atributo de controle para saber se esse *changelog* está publicado.
software	*Software* ao qual este *changelog* pertence.
description	Descrição do *changelog*.

Fonte: Autor.

Método:

***verifyPendingChangelogs(Changelog changelog)*:** É chamado quando um membro vai publicar o *changelog*, válida se existem changelogs criados antes deste antes de publicar o *changelog* passado como parâmetro, pois compreende-se que se existe um changelog posterior não publicado mudanças ficaram de fora do histórico.

Entry

Entry é a classe representação de uma entrada ou mudança feita no *software*.

Tabela 5 - Atributos da classe *Entry*.

Atributo	Descrição
id	Identificador único.
title	Título da mudança.
description	Descrição desta mudança.
type	É um *enum*: *EntryType*, que identifica o tipo desta entrada.
resource	Recurso do *software* ao qual essa entrada pertence.
changelog	Recurso ao qual a mudança compreende.

Fonte: Autor.

Tabela 6 - Valores do *Enum EntryType*, referente ao atributo *type*.

Nome	Descrição
ADDED	Indica que a essa mudança se refere a uma funcionalidade adicionada ao sistema.
CHANGED	Funcionalidade, recurso alterado.
DEPRECATED	Funcionalidade como, por exemplo, um método que no futuro será excluído do sistema.
REMOVED	Funcionalidade que por hora foi removida do sistema.

FIXED	Funcionalidade arrumada indica uma correção.
SECURITY	Utilizada no caso de mudanças que envolvem a segurança do sistema.

Fonte: Autor.

Esse *enum* foi criado utilizando os tipos de mudanças descritas no *Keep a Changelog*. Disponível em <https://keepachangelog.com/en/1.0.0/>. Acesso em: 26 ago. 2018.

Resource

Representa um recurso do sistema, como por exemplo: cadastro de *software*.

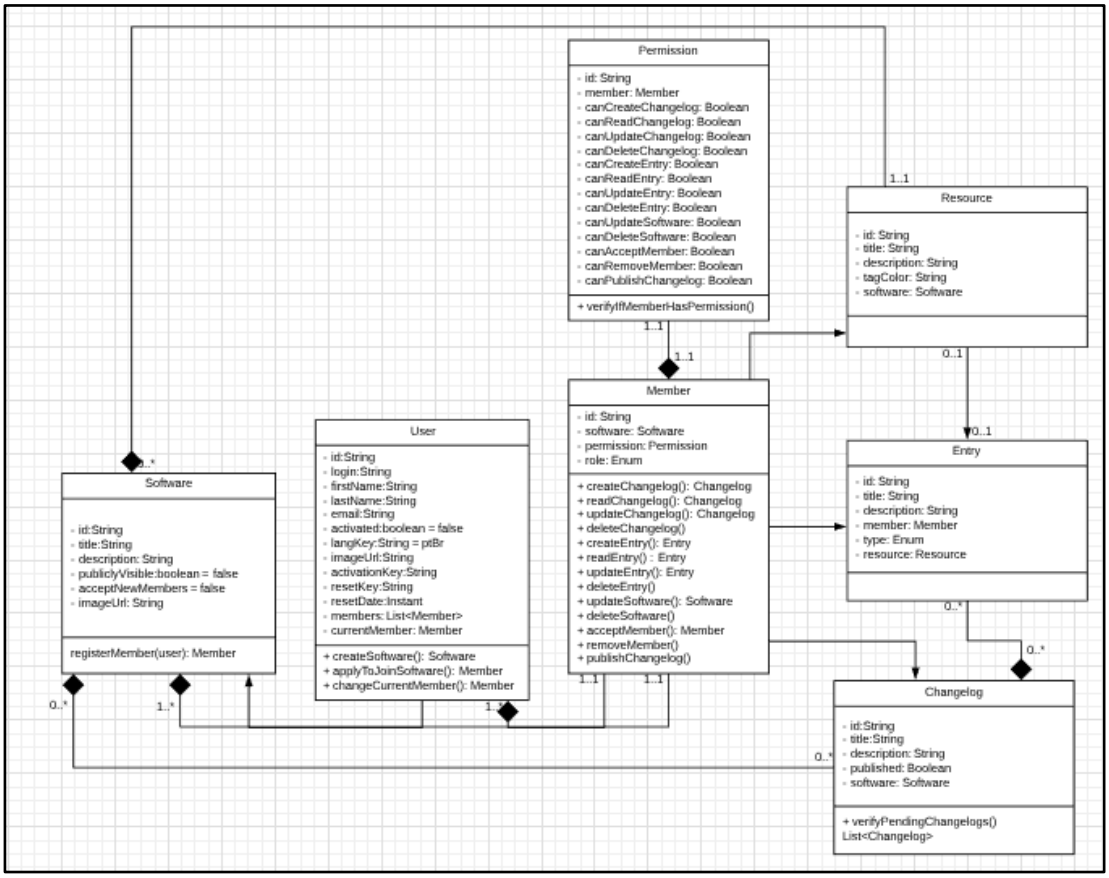
Tabela 7 - Atributos da classe *Resource*.

Atributo	Descrição
id	Identificador.
title	Título do recurso.
description	Descrição do recurso.
tagColor	Código hexadecimal de uma cor. É utilizado para facilitar a visualização.
software	*Software* à que esse recurso pertence.

Fonte: Autor.

Juntando todos os módulos, no final chega-se a seguinte representação do sistema apresentada na figura 9:

Figura 9 - Diagrama de classe com todas as classes do sistema.

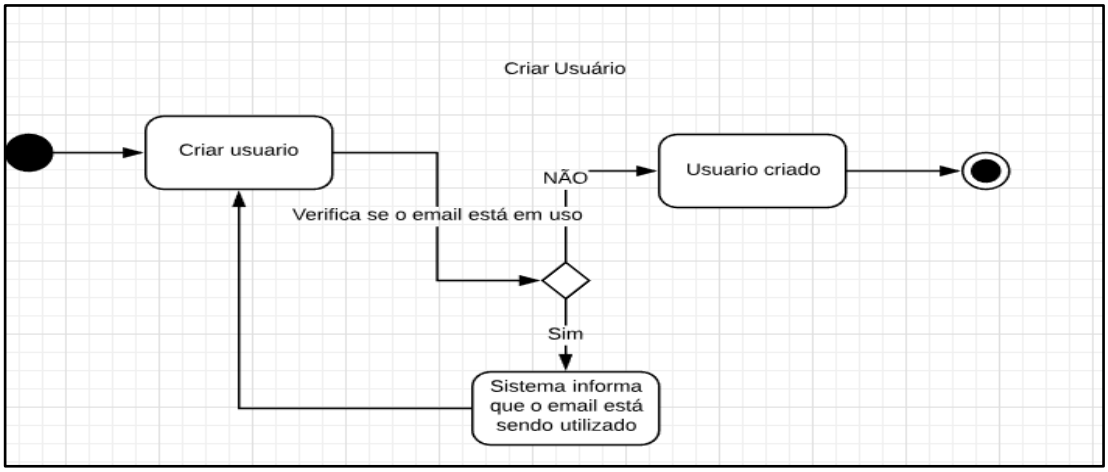


Fonte: Autor do trabalho.

3.3 DIAGRAMA DE ATIVIDADES

Como descrito no capítulo de engenharia de *software*, o diagrama de atividades serve para representar a cadeia de ações dentro de uma operação do sistema. Para o sistema deste trabalho, é então dada a representação das principais operações que podem ser executadas dentro do sistema.

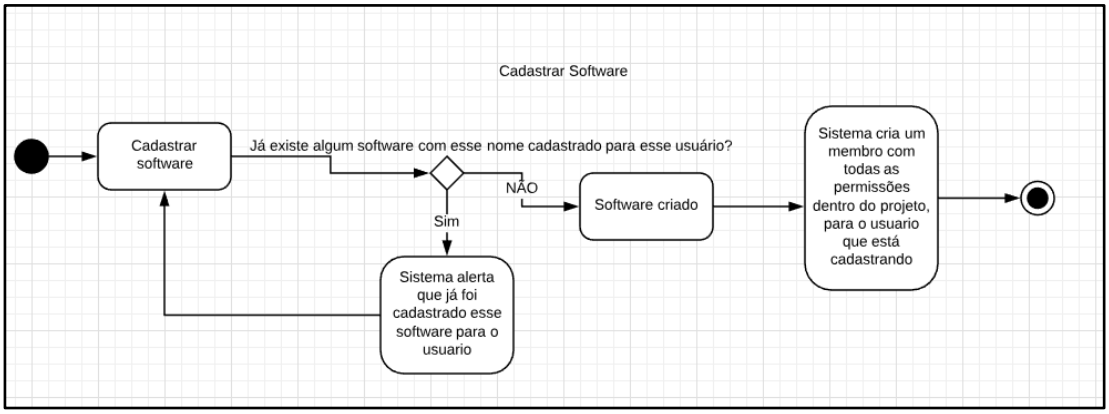
Figura 10 - Diagrama de atividade - criar usuário.



Fonte: Autor.

O diagrama da figura 10 representa a atividade de cadastrar um usuário no sistema, é possível descrevê-lo da seguinte forma: uma pessoa inicia o processo de criar um usuário, representado pela ação criar usuário, nesta etapa a pessoa preenche um formulário com os dados necessários, apresentados na classe *User*, ao enviar (confirmar) esse formulário o sistema faz a validação se o e-mail já está cadastrado para outro usuário, se estiver avisa a pessoa que o e-mail já está em uso, a partir disso ele pode tentar cadastrar com outro e-mail ou tentar recuperar sua senha. Caso não existe um usuário cadastrado com o e-mail informado o sistema cria o usuário.

Figura 11 - Diagrama de atividade - cadastrar *software*.

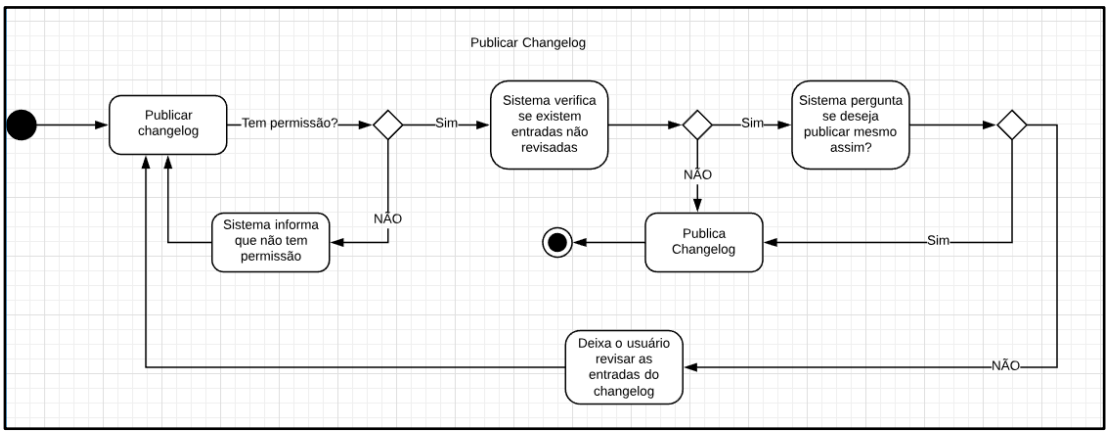


Fonte: Autor.

O diagrama da figura 11 representa o processo de cadastrar um *software* no sistema, similar a atividade de criar um usuário é verificado se já existe um *software* com o nome

(atributo da classe *Software*) já cadastrado para o usuário que está efetuando a operação, após o sistema criar o *software* (registrar ele no banco), o mesmo cria um membro dentro deste *software* para o usuário, esse membro já é criado com todas as permissões habilitadas. Isto é, neste processo são gerados três registros *software*, *member* e *permission*.

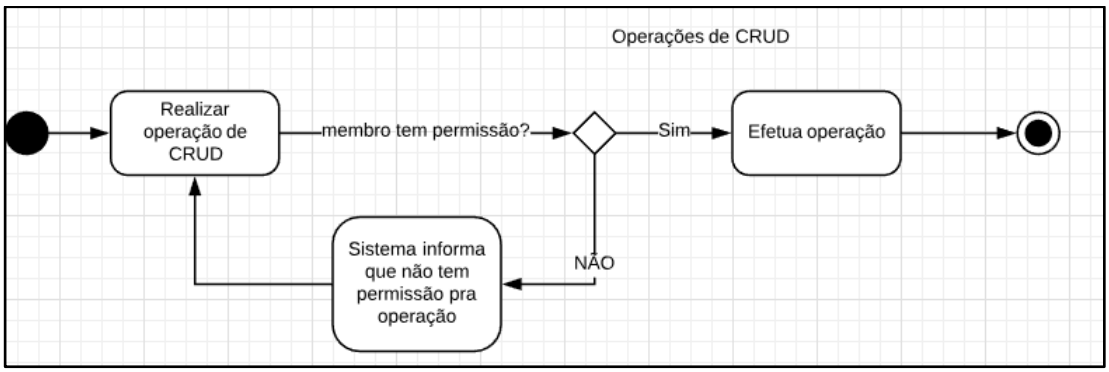
Figura 12 - Diagrama de atividade - publicar *changelog*.



Fonte: Autor.

O diagrama da figura 12 representa o processo de publicar um changelog, que é o mais complexo do sistema. Quando um membro inicia o processo de publicar o *changelog*, é verificado se o mesmo tem permissão para esta operação, caso tenha o sistema verifica se existem entradas não revisadas no changelog que se quer publicar, caso todas as entradas tenham sido revisadas o sistema prossegue para publicar o *changelog*, do contrário, pergunta se o membro deseja publicar mesmo assim, caso o membro selecione não, é aberta uma tela para ele revisar as entradas, feito isso a atividade volta para a ação de publicar o *changelog*. Isto porque, no meio tempo em que ele está revisando as entradas pode ocorrer de ele perder a permissão para publicar o changelog, então essa permissão é verificada novamente. A atividade encerra com o *changelog* sendo publicado.

Figura 13 - Diagrama de atividade - operações de *CRUD*.



Fonte: Autor.

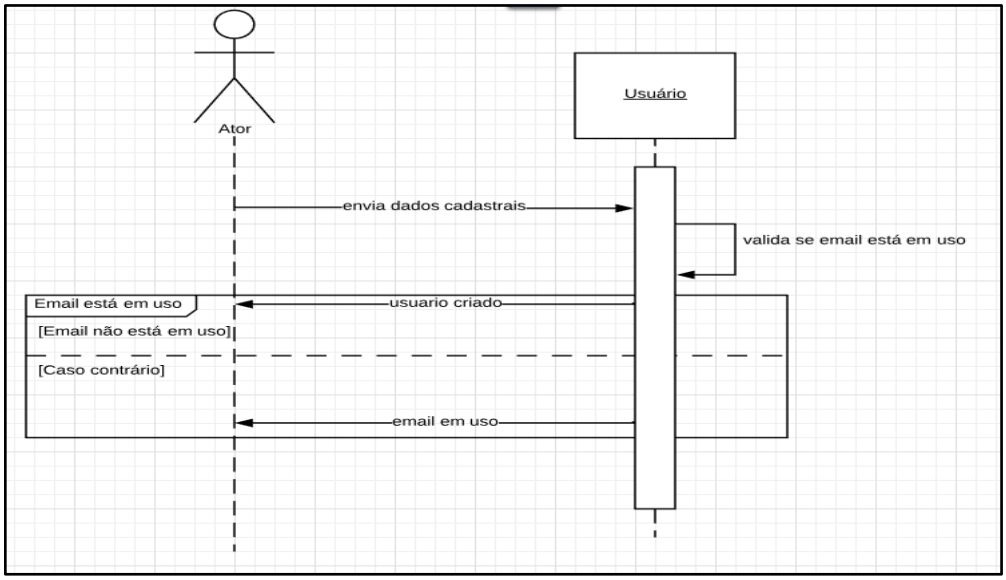
Este último diagrama da figura 13 representa todas as atividades de *CRUD* que um membro pode realizar, como por exemplo, criar, editar, deletar, visualizar um *changelog*, entrada, recurso, *software* ou permissão. Para qualquer operação citada é verificado se o membro tem permissão, se sim efetua a operação, se não apresenta uma mensagem dizendo o mesmo não possui a permissão para realizar a operação.

3.4 DIAGRAMA DE SEQUÊNCIA

Como introduzido no capítulo de engenharia de *software*, o diagrama de sequência serve para descrever a sequência em que a troca de mensagens que ocorrem entre os objetos do sistema dentro de uma determinada operação, os diagramas de sequência a seguir descrevem essas trocas usando os mesmos processos utilizados no diagrama de atividades.

O diagrama de sequência da figura 14 representa o processo que um indivíduo inicia ao criar um usuário. Onde é validado se o *e-mail* está disponível para uso.

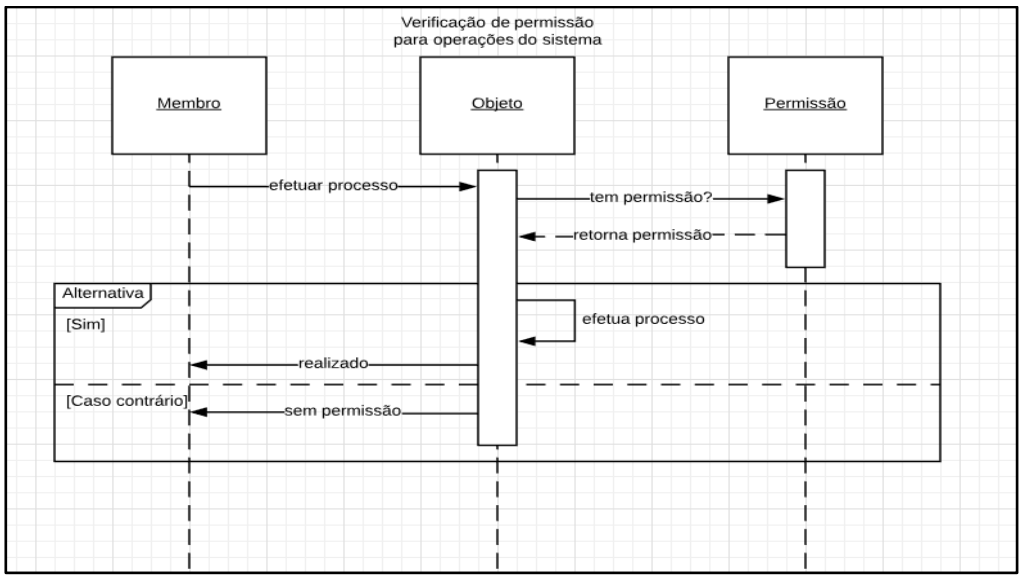
Figura 14 - Diagrama de sequência - criar usuário.



Fonte: Autor.

O diagrama da figura 15 representa a validação que é feita sobre as permissões de um membro, quando este tenta realizar uma operação dentro dos objetos pertinentes ao *software* em que é membro. É usada a palavra operação para representar qualquer processo que o usuário irá realizar, como as operações de *CRUD*. No diagrama é utilizado um objeto com nome Objeto para abstrair qualquer objeto do sistema que tenha permissionamento.

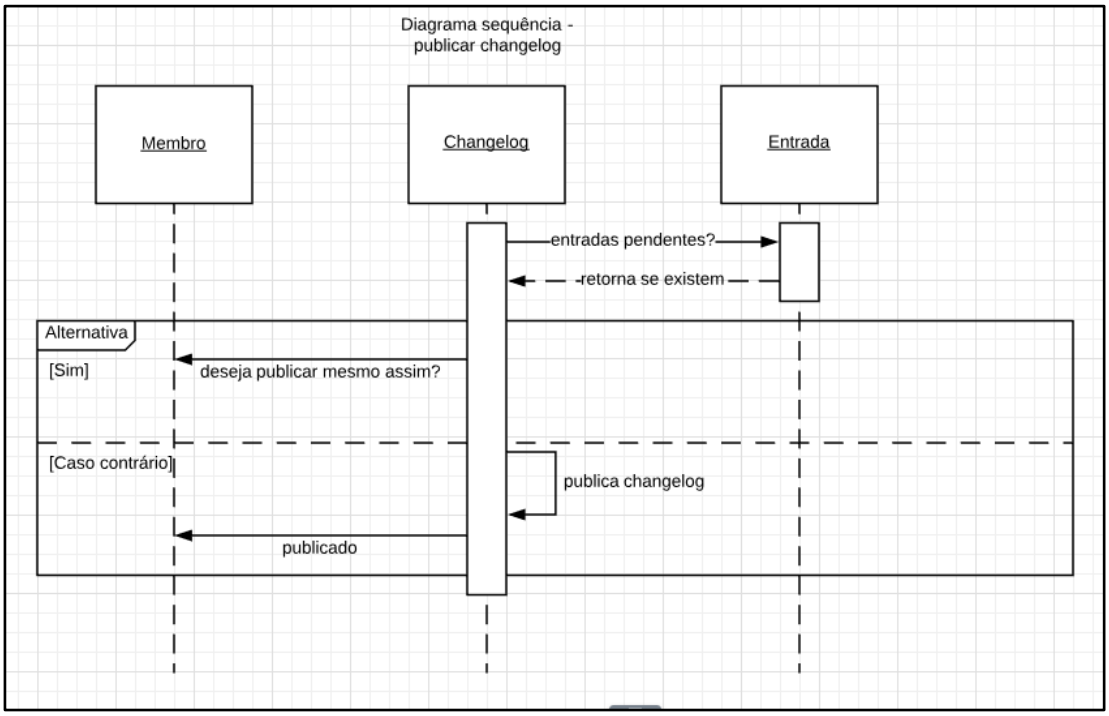
Figura 15 - Diagrama de sequência - verificação de permissão.



Fonte: Autor.

Partindo da sequência apresentada pelo diagrama da figura 15, quando um membro vai publicar um changelog é realizada a seguinte sequência apresentada na figura 16.

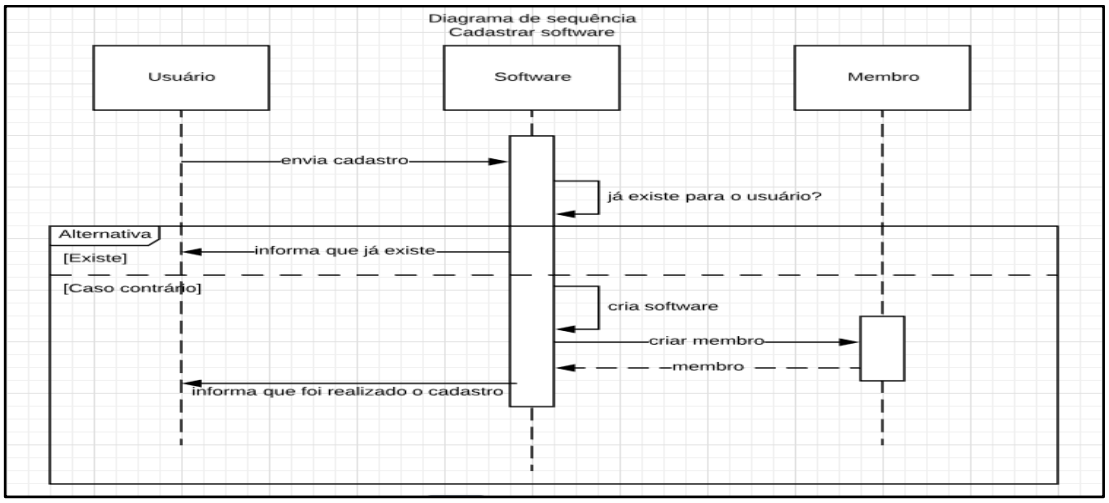
Figura 16 - Diagrama de sequência - publicar changelog.



Fonte: Autor.

Por último na figura 17 é apresentado o diagrama de sequências para o cadastro de *softwares*.

Figura 17 - Diagrama de sequência - cadastrar *software*.



Fonte: Autor.

4. SISTEMA *CHANGELOG*

Para o desenvolvimento do *software* proposto por este trabalho, será utilizada a plataforma de desenvolvimento *JHipster*. Esta ferramenta unifica alguns *Frameworks* de desenvolvimento e serve basicamente para gerar código a partir das entidades (classes) especificadas, disponibilizando uma estrutura inicial para o sistema.

O conceito por trás desta ferramenta é fornecer convenção sobre configuração, que é um modelo de desenvolvimento de *software* que busca diminuir o número de decisões que desenvolvedores precisam tomar, tornando o processo de desenvolvimento mais simples e focado nas funcionalidades mais particulares do negócio (regras de negócio).

Como dito o *JHipster* unifica alguns *Frameworks* de desenvolvimento e banco de dados, para gerar uma aplicação inicial configurada e pronta para implementação. A seguir são apresentados alguns dos principais *frameworks* que compõem o sistema, e são utilizados no desenvolvimento deste trabalho.

O *Framework Spring Boot* é um projeto *open source* criado para facilitar a tarefa de se criar uma aplicação do zero, isto porque ela se encarrega de fazer toda a configuração inicial do sistema seguindo também o conceito de convenção sobre configuração, provendo uma forma rápida de se ter uma aplicação executando, além disso, ela também oferece algumas funcionalidades não funcionais como servidor embutido, ferramentas de métricas, status do sistema e segurança. Sua documentação está disponível em: https://docs.spring.io/spring-boot/docs/current-SNAPSHOT/reference/html/.

O *Angular*, plataforma para *front-end* mantido pela *Google*, é utilizado no desenvolvimento de aplicações *web cross-platform*, isto é aplicações que executam em ambientes *web*, móveis ou *desktop*. Aplicações criadas em angular são *SPA* (*single-page-application*). Uma *single page application* carrega todos os recursos em um único carregamento, ou vai carregando-os conforme requisitadas. Sua documentação pode ser encontrada no site: https://angular.io/docs.

Para o banco de dados do sistema que será desenvolvido é utilizado o *MongoDB*, este é um banco *no-sql* orientado a documentos. Em seu site pode encontrar: *MongoDB* armazena dados em documentos flexíveis similares a *JSON* significando que os campos de um documento podem variar de documento para documento e que a estrutura pode facilmente ser modificada com o tempo (Disponível em: <https://www.mongodb.com/what-is-mongodb>, Acesso em: 28, ago.2018). Com outras palavras um documento do *MongoDB* pode ser análogo à uma tupla de um Banco de Dados Relacional, diferindo apenas no formato e estrutura do armazenamento. Como a estrutura dos documentos é similar a de um *JSON* que por sua vez é similar a de uma classe *Java*, foi escolhido o *MongoDB*, pois este vai possibilitar que o Banco de Dados siga a mesma estrutura apresentada do diagrama de classes sem que seja necessário traduzir toda a estrutura para *SQL*. A documentação do *MongoDB* pode ser acessado no site: https://docs.mongodb.com/.

É claro que essas plataformas ainda possuem muitas outras funcionalidades e outros *Frameworks* ou plataformas incorporadas, a ideia aqui foi apenas apresentar brevemente quais são as principais ferramentas que serão utilizadas neste trabalho.

4.1 GERANDO A APLICAÇÃO

Utilizando o *JHipster* é possível gerar uma aplicação base, na qual o sistema proposto por este trabalho será construída. A partir de um arquivo *JSON rc-yo.json* são declaradas as opções que serão utilizadas para gerar a aplicação base ou casco, como por exemplo o banco de dados que vai ser utilizado, o *Framework* do *front-end*, tipo de autenticação, *Framework* de testes, etc. O arquivo pode ser encontrado nos apêndice A deste trabalho.

Dentro do diretório de onde o arquivo está, executa-se o comando *JHipster*, que vai fazer com que o sistema inicial seja gerado. Essa aplicação inicial já disponibiliza os arquivos fonte básicos de *front-end* e *back-end* e é sobre essa aplicação inicial ou casco que o sistema proposto por esse trabalho irá ser desenvolvido.

4.2 GERAÇÃO DAS CLASSES OU ENTIDADES

Para gerar as classes que foram apresentadas nos diagramas do Capítulo 3, utiliza-se uma ferramenta *web* que o *JHipster* fornece, o *JDL-Studio*. Esta ferramenta serve para

descrever as entidades (classes) e relacionamentos que formam o sistema. A partir dela é possível gerar um diagrama *UML* e o arquivo .jh que será importado no projeto para gerar as classes, relacionamentos e *Enums* do sistema. Ela abrange quase todos os tipos de variáveis *Java* e relacionamentos de banco como um-para-muitos, um-para-um e muitos-para-muitos. Mais detalhes sobre a ferramenta podem ser encontrados no site https://www.jhipster.tech/jdl/.

A ferramenta em si pode ser acessada do *link* https://start.jhipster.tech/jdl-studio/, quando carregado pela primeira vez no navegador o *JDL-Studio* vai trazer um diagrama exemplo, demonstrando suas funcionalidades declarando algumas entidades, *Enums* e relacionamentos. Um relacionamento é declarado no *JDL* como:

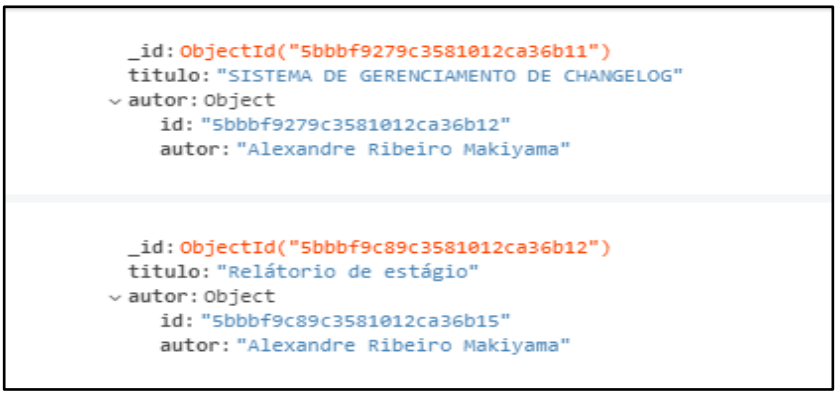
Figura 18 - Declaração de relacionamento *JDL*.

```
entity Book
entity Author

relationship ManyToOne {
  Book to Author
}
```

Fonte: https://www.jhipster.tech/jdl/, Acesso em: outubro de 2018.

Na figura 18 é declarado um relacionamento entre duas entidades livro e autor, no caso o relacionamento criado é de muitos livros para um autor, quando utilizado um Banco Relacional *SQL* este relacionamento será representado com uma chave estrangeira do autor na tabela de livros. Porém o banco escolhido para a aplicação é o *MongoDB*, um banco *no-sql*, no qual o armazenamento dos dados ocorre na forma de arquivos *BSON*, capazes de armazenar o relacionamento com o objeto inteiro, neste sentido então dentro de cada existe um objeto autor, como na figura 19.

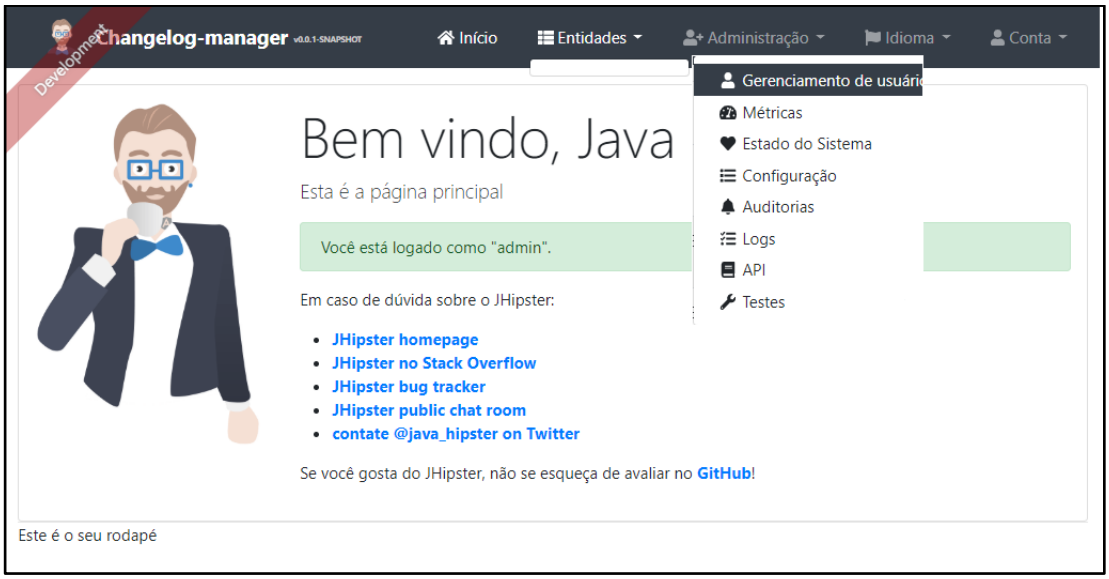
Figura 19 - Exemplo - Relacionamento *MongoDB*.



Fonte: Autor.

O sistema já é gerado com telas de auditoria do sistema e um módulo de usuários. A figura 20 ilustra uma aplicação recém-criada, utilizando somente o arquivo rc-yo.json dado como exemplo, e como se pode ver o menu de entidades está vazio, porque não foi passado nenhum arquivo .jh durante a geração da aplicação. Desta forma as únicas entidades que foram criadas são a User e as entidades de auditoria, para elas também foram geradas telas que podem ser acessadas no menu administração.

Figura 20 - Tela principal de uma aplicação recém gerada.

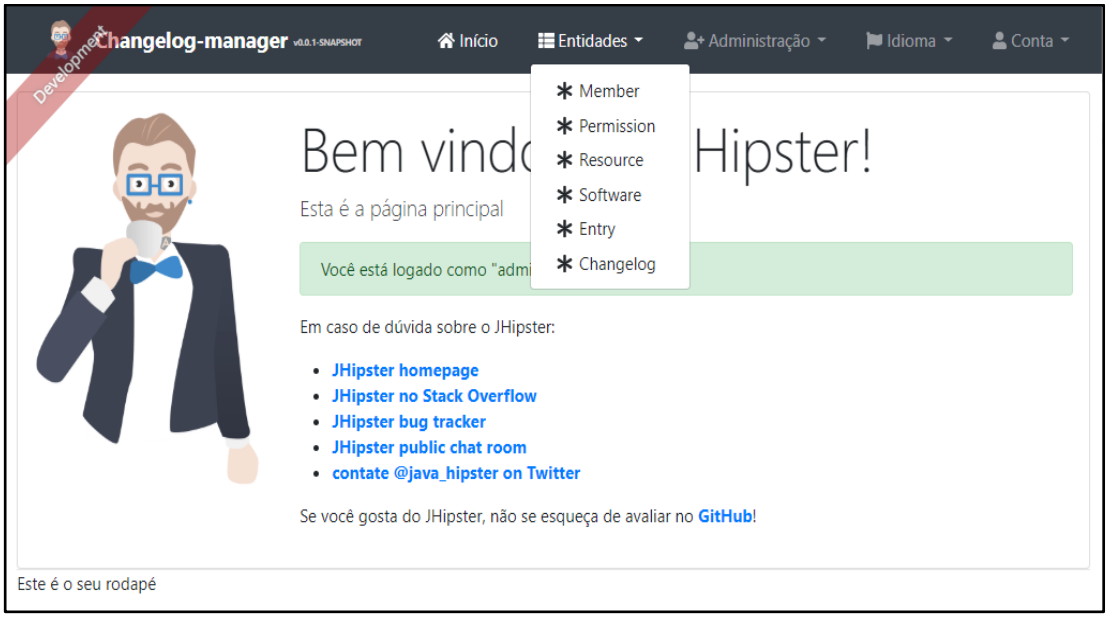


Fonte: autor.

Partindo da aplicação gerada é então importado as entidades especificadas nos arquivos *member-jdl.jh*, *permission-jdl.jh*, *resource-jdl.jh*, *entry-jdl.jh*, *changelog-jdl.jh*

disponibilizados no apêndice B deste trabalho. Após rodar o comando de importação o menu de entidades é atualizado com as entidades geradas. Como demonstrado na figura 21.

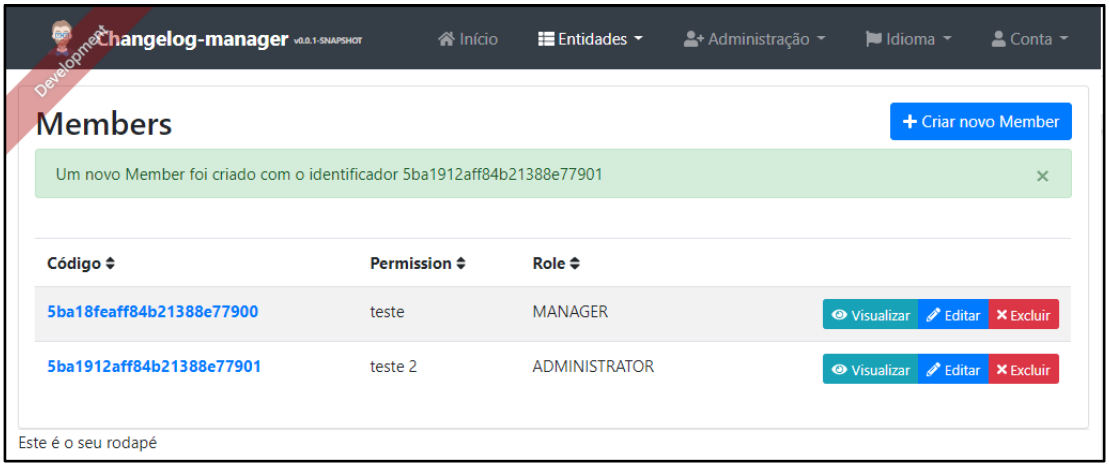
Figura 21 - Menu com as entidades geradas.



Fonte: Autor.

A partir desses menus é possível listar os registros das entidades que existem no banco, criar, editar e deletar. Ao importar a entidade foi então gerada uma pequena estrutura que permite fazer basicamente um *CRUD* da entidade gerada como pode ser visualizado na figura 22.

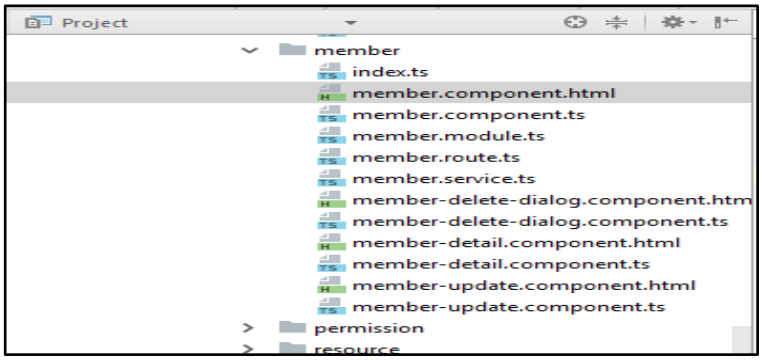
Figura 22 - Exemplo de tela gerada para entidade *Member*.



Fonte: Autor.

A figura 22 também mostra uma lista com dois membros cadastrados como exemplo, esta tela gerada lista todos os membros criados no sistema que estão na coleção de dados *Members*, através dela também é possível fazer a visualização detalhada do registro, cadastro de novos registros, edição de registro ou apaga-los, às telas dessas operações citadas estão no apêndice C do trabalho. A estrutura de arquivos gerada para cada entidade do *front-end* pode ser visualizada na figura 23.

Figura 23 - Exemplo de estrutura gerada para o *front-end*.



Fonte: Autor.

Os arquivos são referentes às telas criadas para a entidade *Member*, sendo compostas por uma *view*: arquivo *.html*, que é a tela, um *controller*: controlador da *view*, serve para controlar os componentes que estão em tela, um *route*: mantém as rotas de navegação entre os componentes, um *module*: classifica e agrupa os componentes do módulo e um *service*: contém e faz requisições para o *back-end* realizar processos.

No *back-end* uma estrutura similar é gerada, um arquivo *resource*: responsável por receber as requisições e retornar as respostas para o *front*, *service*: processa a requisição e *repository*: processa as operações de banco de dados. Esses arquivos nomeados utilizando o padrão: nome da entidade mais o tipo de controle, então para membro, por exemplo, foram gerados: *MemberResource.java*, *MemberService.java* e *MemberRepository.java* além é claro do próprio *Member.java* que é classe utilizada para representar os membros.

Tomando como exemplo o processo de cadastro de *software* que qualquer usuário pode fazer. O mesmo acessa uma tela *software software.component.html* e inicia um cadastro, ao clicar no botão salvar o *controller* desta *view* faz a validação básica dos dados que foram inseridos no formulário e chama a função de salvar no *software.service.ts*, que dispara uma

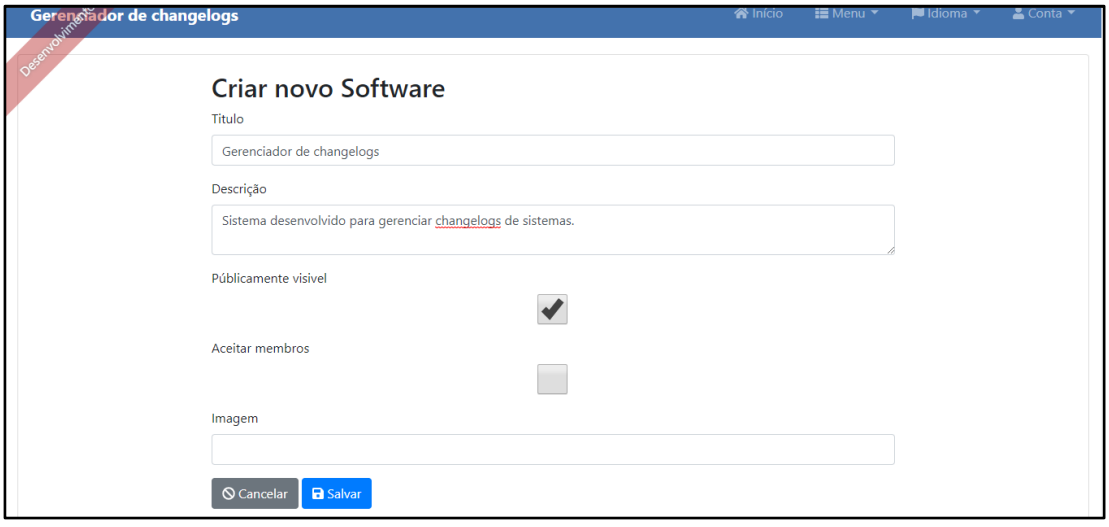
requisição no *SoftwareResource*, recebendo essa requisição, é então chamado o método responsável pela criação do registro na classe *SoftwareService*, onde é feita a validação de nome mostrada na figura 17, que utiliza o *SoftwareRepository* para fazer a verificação e criar ou não o registro de *software*. No final do processamento é retornada a mensagem em tela para o usuário, como na figura 22, aonde é realizado o cadastro de membro com sucesso.

4.3 DESENVOLVIMENTO

Agora com todas as entidades especificadas devidamente importadas para a aplicação, é preciso desenvolver as regras do sistema e um novo *layout* para a aplicação não ficar com a mesma aparência de todas as outras geradas pelo *Jhipster*. Neste momento então já é possível criar registros no banco, porém sem nenhuma regra, isto é, não existem os processos, descritos na especificação e o usuário pode ir simplesmente cadastrando e gerenciando objetos no banco como desejar.

O desenvolvimento será iniciado pela rotina de cadastrar *softwares*, que a partir da tela da figura 24, vai enviar os dados inseridos pelo usuário para o servidor que automaticamente irá criar seu membro de projeto com suas permissões.

Figura 24 - Tela cadastro de *software*.



Fonte: Autor.

Ao clicar no botão salvar, os dados são enviados para a *API* da figura 25, onde é verificado no método *doesSoftwareAlreadyExistsForUser(software)* se o usuário já não tem um *software* com o título informado, se for verificado que existe, é retornado um erro.

Figura 25 - API que recebe o objeto de *Software*.

```
@PostMapping("/softwares")
@Timed
public ResponseEntity<Software> createSoftware(@RequestBody Software software) throws URISyntaxException {
    log.debug("REST request to save Software : {}", software); // recebe um objeto software

    if (softwareService.doesSoftwareAlreadyExistsForUser(software)) { // validação se o usuario já tem
        // um software com esse mesmo nome cadastrado
        throw new BadRequestAlertException("There's already a software with this title", ENTITY_NAME, "titleexists");
    }

    final Software result = softwareService.create(software);
    return ResponseEntity.created(new URI( str: "/api/softwares/" + result.getId()))
        .headers(HeaderUtil.createEntityCreationAlert(ENTITY_NAME, result.getId()))
        .body(result); // retorna o software cadastrado para o front
}
```

Fonte: Autor.

Feita a validação o método *create* mostrado na figura 26 salva o cadastro no banco e chama o serviço de membro para criar um novo membro e suas permissões para o projeto que acabou de ser salvo.

Figura 26 - Método que salva o cadastro de *software*.

```
public Software create(final Software software) {
    log.debug("Request to create Software : {}", software);

    final Software savedSoftware = softwareRepository.save(software);
    memberService.create(savedSoftware, permissionService.createWithFullPermission());

    return savedSoftware;
}
```

Fonte: Autor.

O *create* do serviço de membros exibido na figura 27, então atribui os valores no objeto de membros, e retorna o *software* cadastrado para o usuário, informando que o cadastro foi finalizado com sucesso.

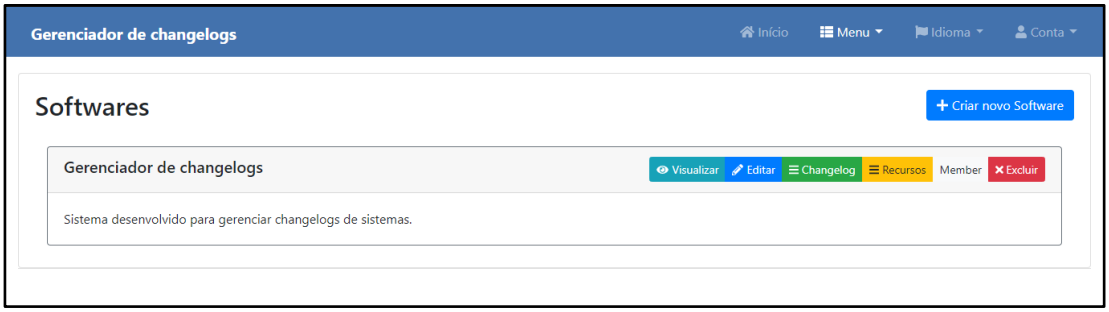
Figura 27 - Método que salva o membro e permissão.

```
public Member create(final Software software, final Permission permission) {
    log.debug("Request to create Member in software {}, with permission {}", software, permission);

    final Member member = new Member();

    member.setSoftware(software.getId());
    member.setPermission(permission.getId());
    member.setUser(software.getUserId());
    member.setRole(MemberRole.ADMINISTRATOR);

    return memberRepository.save(member);
}
```

Fonte: Autor.

Após executar a rotina desenvolvida, o *software* criado é então listado para o usuário que o criou na tela da figura 28, onde é possível acessar as outras funcionalidades e gerenciar o sistema:

Figura 28 - Tela lista de softwares cadastrados.

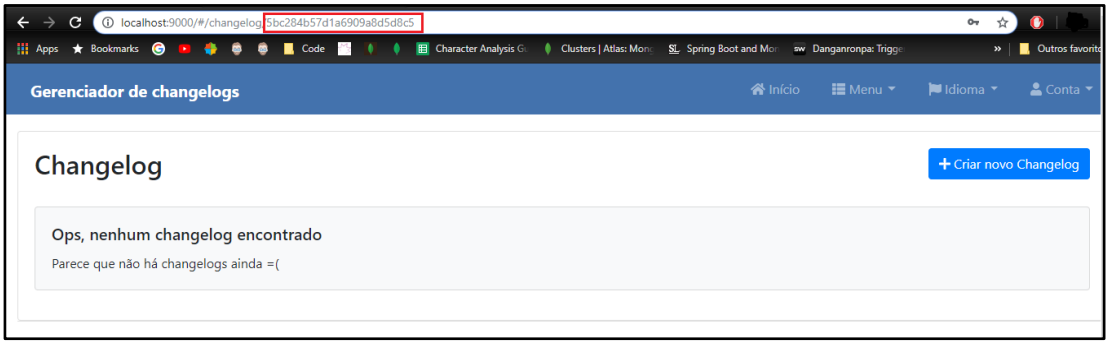


Fonte: Autor.

Para essa tela o *layout* já foi alterado, ao invés de utilizar uma tabela *HTML*, foi elaborado um novo *layout* usando um estilo de *card*, que é disponibilizado pelo *Bootstrap*. Todas as telas de gerenciamento, isto é, as telas que fazem operações de inserção, modificação, remoção ou visualização vão seguir o padrão da tela acima.

Nesta etapa do desenvolvimento também foram determinadas as rotas de navegação do sistema, porque as entidades são ligadas uma nas outras, como mostrado nos diagramas de classe, por exemplo: um objeto *changelog* tem um atributo *software*, que é o *id* do *software* ao qual o *changelog* pertence. Então ao clicar no botão *changelog*, o *id* do *software* é passado para a próxima tela, através da *url*, como mostrado na figura 29.

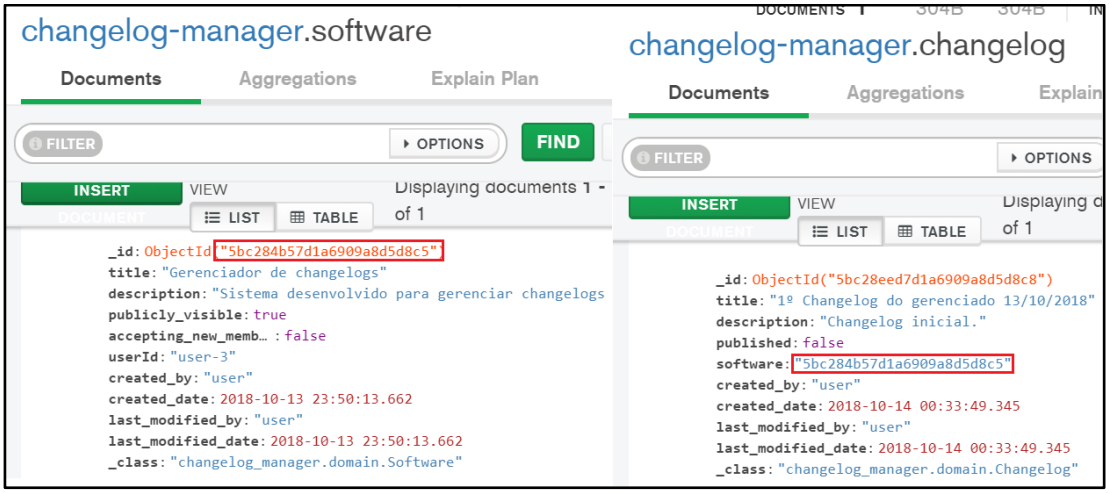
Figura 29 - Passagem do *id* de *software* para a tela de *changelog*.



Fonte: Autor.

O valor do *id* é então atribuído a uma variável no componente da tela, e quando um *changelog* vai ser criado essa variável é utilizada para preencher o atributo *software* do *changelog*, que será salvo no banco, como mostrado na figura 30, dos dados já salvos no banco de dados.

Figura 30 - Registro de *software* e changelog.



Fonte: Autor.

Para todas as demais entidades que carregam referência foi desenvolvido o mesmo processo, alterando somente a referência que é salva, com isso foram então desenvolvidas as consultas para recuperar apenas os registros pertinentes ao *id* referencia passado, por exemplo: ao abrir a tela de *changelogs*, através da tela de *softwares*, somente os *changelogs* que carregam a referência passada serão exibidos em tela, impedindo que se misturem *changelogs* de outros *softwares*.

Figura 31 - API para consulta de *changelogs*.

```
36        querySoftwareChangelogs(softwareId: string, req?: any): Observable<EntityArrayResponseType> {
37            const options = createRequestOption(req);
38            return this.http.get<IChangelog[]>( url: `${this.querySoftwareChangelogsUrl}/${softwareId}`,  options: { params:
39        }
```

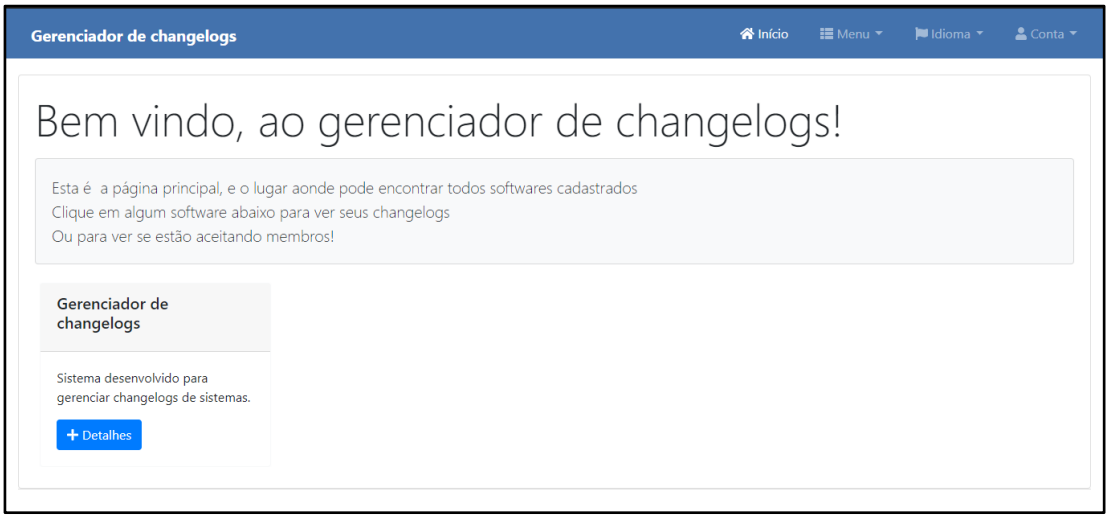
```
@GetMapping("/software-changelogs/{softwareId}")
@Timed
public ResponseEntity<List<Changelog>> getAllSoftwareChangelogs(Pageable pageable,
                                                                @PathVariable final String softwareId) {
    log.debug("REST request to get a page of Changelogs");

    // retorna uma uma de changelogs que tem a referencia do software passado
    Page<Changelog> page = changelogRepository.findAllBySoftware(softwareId, pageable);
    HttpHeaders headers = PaginationUtil.generatePaginationHttpHeaders(page,  baseUrl: "/api/software-changelogs")
    return new ResponseEntity<>(page.getContent(), headers, HttpStatus.OK);
```

Fonte: Autor.

A mesma lógica da *API* da figura 31 é utilizada para a listagem das outras entidades que têm referência de outro objeto.

Com todos os processos de *CRUD* para as entidades desenvolvidas, foi alterada a tela inicial para exibir todos os *softwares* que possuem o atributo publicamente visível com valor igual à *true*, como no *software* que foi criado, que agora aparece listado na tela inicial da figura 32, onde é possível que qualquer usuário veja mais detalhes sobre o *software*.

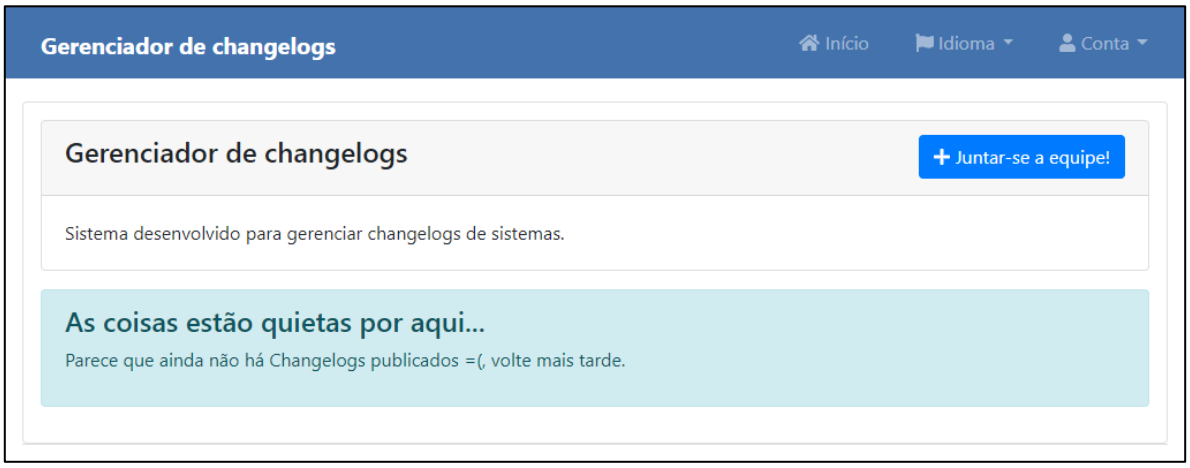
Figura 32 – Tela inicial.



Fonte: Autor.

Ao clicar em mais detalhes o usuário é redirecionado para a tela de *changelogs*, que exibe o nome do *software*, sua descrição e seus *changelogs*. Como o *software* ainda não possui *changelogs* publicados é exibido uma mensagem padrão, indicando que não há *changelogs* ainda. A partir desta tela mostrada na figura 33 também é possível requisitar tornar-se um membro do *software*.

Figura 33 - Tela de visualização de *changelogs* (sem *changelogs* publicados)



Fonte: Autor.

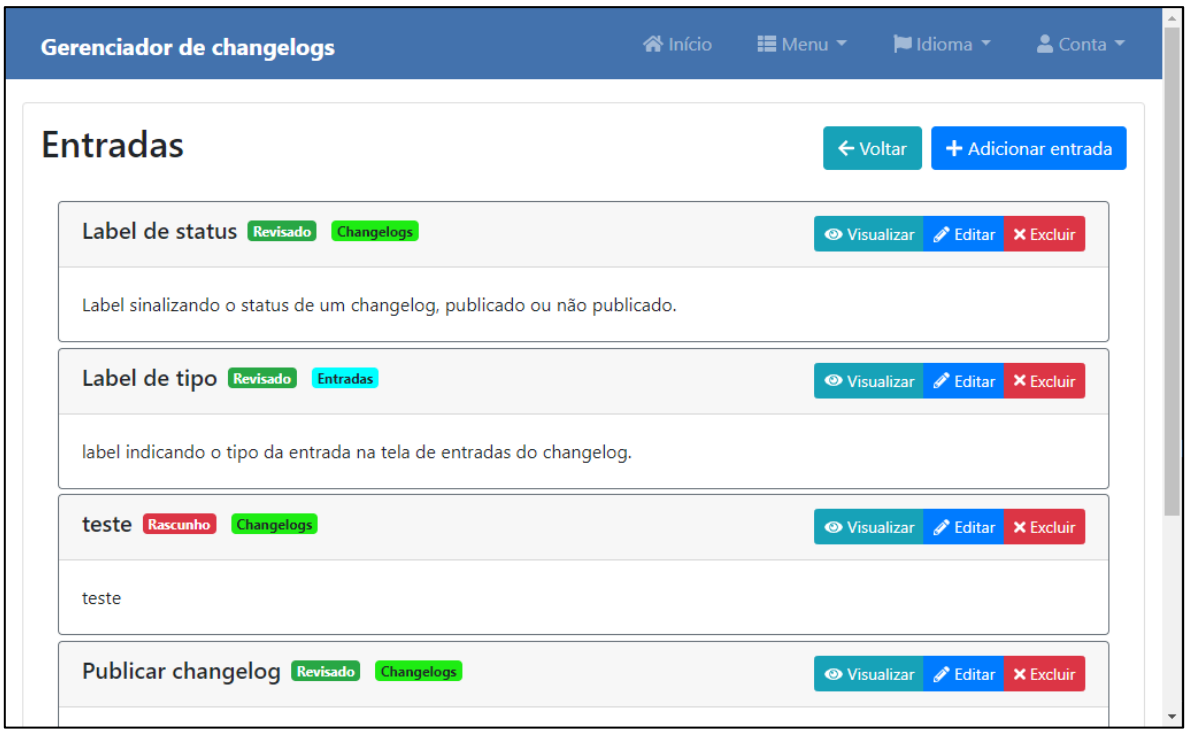
O cadastro de *changelogs* é feito através de sua, acessada através da tela de *softwares*, a próxima imagem mostra a tela com dois *changelogs* cadastrados, onde é também possível acessar a tela de entradas do *changelog* ou publica-lo, como pode ser visto na figura 34.

Figura 34 - Tela de *changelogs*.

Fonte: Autor.

Na tela de entradas mostrada na figura 35 são listadas as entradas que foram adicionadas ao *changelog* selecionado, a imagem abaixo mostra as entradas do *changelog* 31/10/2018. Na tela também é possível fazer o *CRUD* de cada entrada.

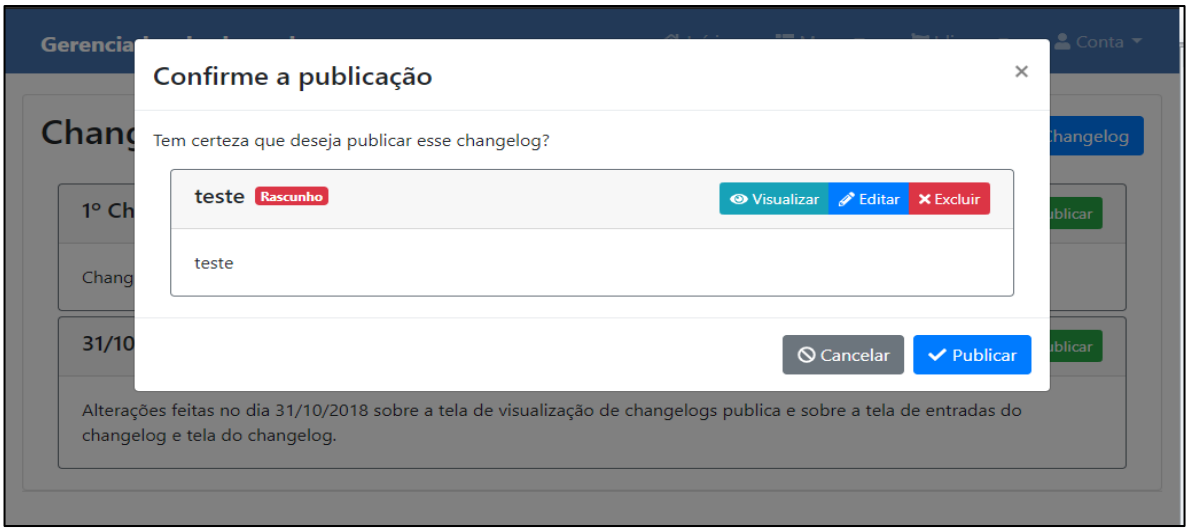
Figura 35 - Tela de entradas.



Fonte: Autor.

Para publicar o *changelog* o usuário vai até a tela de *changelogs* da figura 34 e clica no botão publicar, ao realizar essa ação o sistema faz uma verificação para ver se o *changelog* possui entradas, o *changelog* possuindo pelo menos uma entrada é então feita uma consulta no Banco para recuperar todas as entradas do *changelog* que está sendo publicado.

Figura 36 - Modal de publicação.



Fonte: Autor.

No caso da figura 36, como o *changelog* selecionado para publicação possui uma entrada não revisada, isto é, com o *status* de rascunho, ela é exibida nesta tela para que o usuário possa revisa-la se desejar. Essa validação não trava a publicação, mas uma entrada não revisada não irá ficar disponível para visualização na tela da figura 33. Para trocar o status de rascunho para revisado o usuário precisa clicar no botão editar na tela da imagem 35, ou quando estiver publicando o changelog, tela da figura 36. A requisição feita para recolher as entradas pendentes é mostrada na figura 37.

Figura 37 - API Verificação de entradas pendentes.

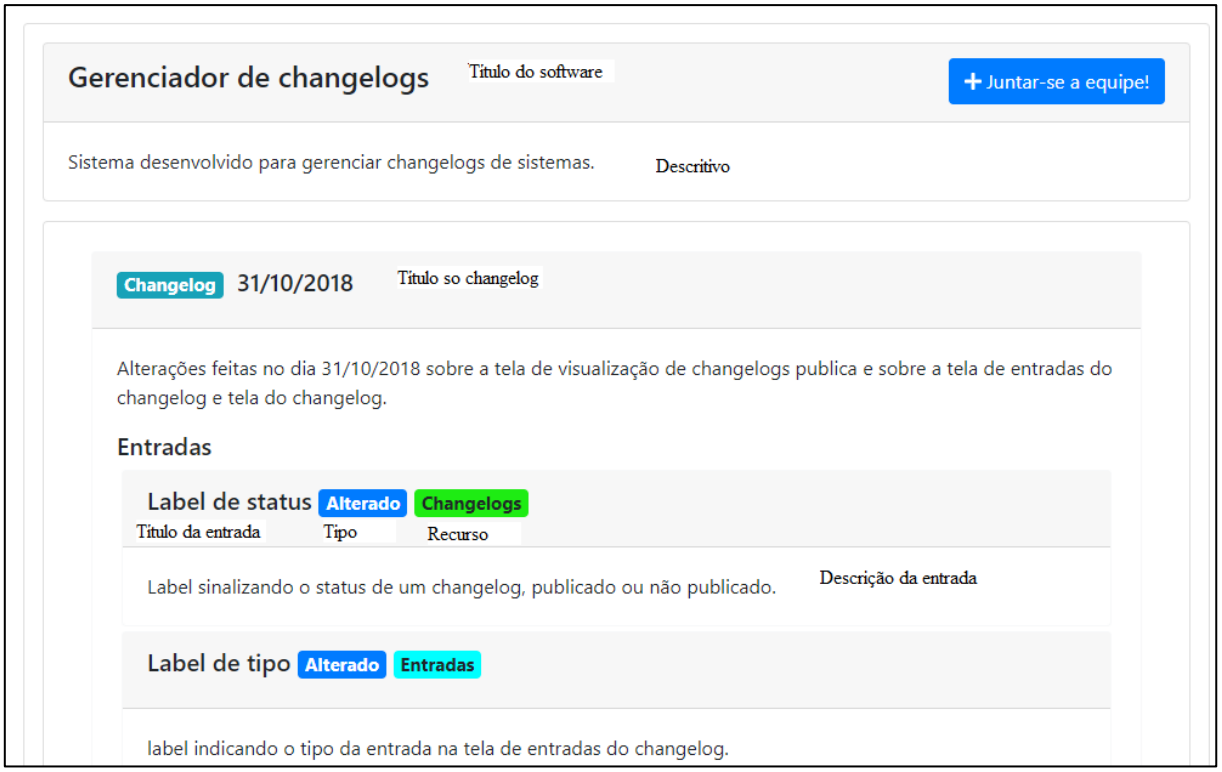
```
@GetMapping("/pending-changelog-entries/{changelogId}")
   @Timed
   public ResponseEntity<?> getAllPendingChangelogEntries(@PathVariable String changelogId) {
       log.debug("REST request to get pending Entries");

       final Optional<Changelog> changelog = changelogService.findOne(changelogId);

       if (changelog.isPresent()) {
           final Optional<List<Entry>> entries = entryService.findAllChangelogEntriesWithStatus(changelogId, EntryStatus.OPEN);

           if (entries.isPresent()) { // if there is pending entries
               return ResponseEntity.badRequest()
                   .headers(HeaderUtil.createEntityAlert(ENTITY_NAME + ".messages", "draftEntriesFound", changelogId)).body(entrie
           } else { // else there is no pending entries
               return new ResponseEntity<>(HttpStatus.OK);
           }
       } else { // if changelog is not found
           return ResponseEntity.notFound()
               .headers(HeaderUtil.createEntityAlert(ENTITY_NAME + ".messages", "changelogNotFound", changelogId)).build();
       }
   }
```

Fonte: Autor.

Na requisição primeiramente é tentado encontrar o *changelog* com o *id* passado, se o sistema não encontrar é retornada uma mensagem dizendo que o *changelog* não existe. Tendo encontrado o *changelog* o sistema pesquisa entradas que precisam ser revisadas, caso não existam o processo segue normalmente e nenhuma entrada é exibida na tela da figura 36. Se houverem entradas que necessitam de revisão elas são exibidas.

Realizado o processo de publicação do *changelog*, o mesmo fica disponível para visualização como na figura 38, onde podem ser visualizadas todas as entradas revisadas do *changelog* que foi publicado.

Figura 38 - Tela de visualização de *changelogs* (com *changelog* publicado)

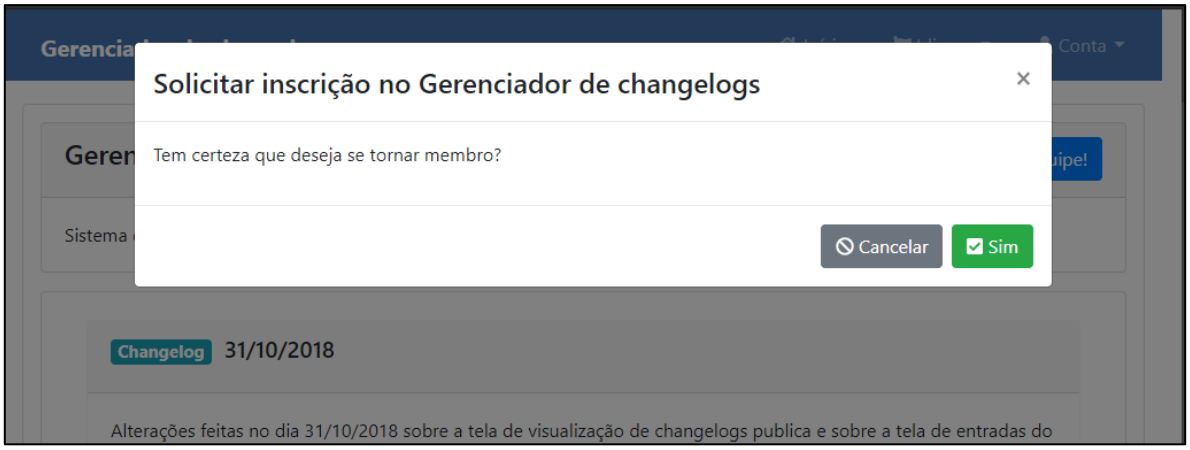


Fonte: Autor.

Concluindo então o desenvolvimento dos recursos que lidam com o gerenciamento dos *changelogs*. Daqui em diante será desenvolvido e apresentado o recurso de gerenciamento de membros dos *softwares* do Gerenciador de c*hangelogs*, este recurso permite mais de um usuário gerenciar os *changelogs* do sistema, podendo a ele ser atribuídas permissões de gerenciamento, como especificado no Capítulo 3.1.

Quando um usuário do sistema deseja se tornar um membro de algum *software*, porque, ele faz parte da mesma equipe de desenvolvimento ou trabalha no mesmo lugar. Ele pode realizar essa requisição na tela da figura 38, no botão juntar-se a equipe. Esse botão abre a caixa de dialogo da figura 39, ao clicar na opção sim é feita uma requisição para criar o membro, esse novo membro sempre é criado sem permissões e com o atributo *accepted* igual à *false*.

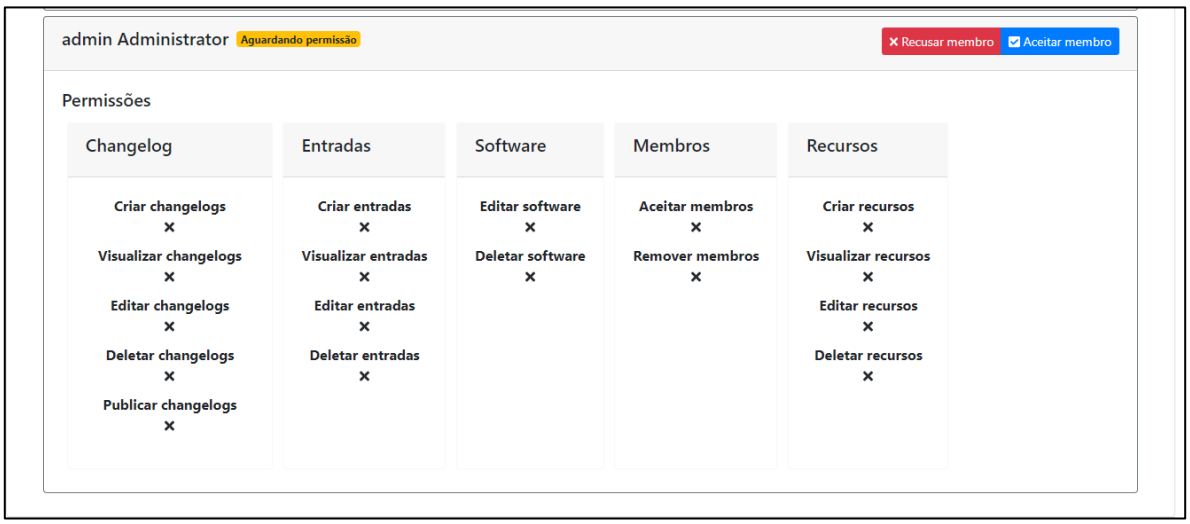
Figura 39 - Solicitação para tornar-se membro.



Fonte: Autor.

Realizado esse processo, o membro criado deverá aparecer na tela de membros do *software* com o estado de aguardando permissão, como a figura 40 ilustra.

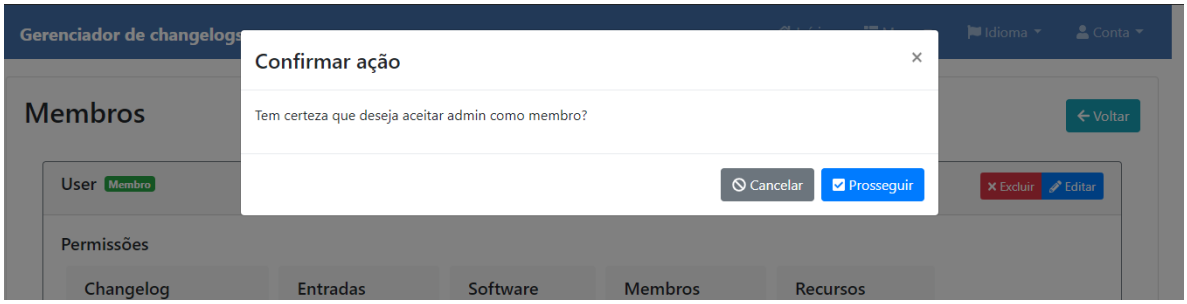
Figura 40 - Membro aguardando aprovação na tela de membros.



Fonte: Autor.

Clicando no botão recusar membro ou aceitar membro, vai ser aberto um dialogo onde é perguntado se realmente deseja-se prosseguir com a ação, caso a opção selecionada tenha sido não, então o membro é excluído, do contrario a solicitação é aceita e o membro aceito poderá realizar as ações que ele tem permissão.

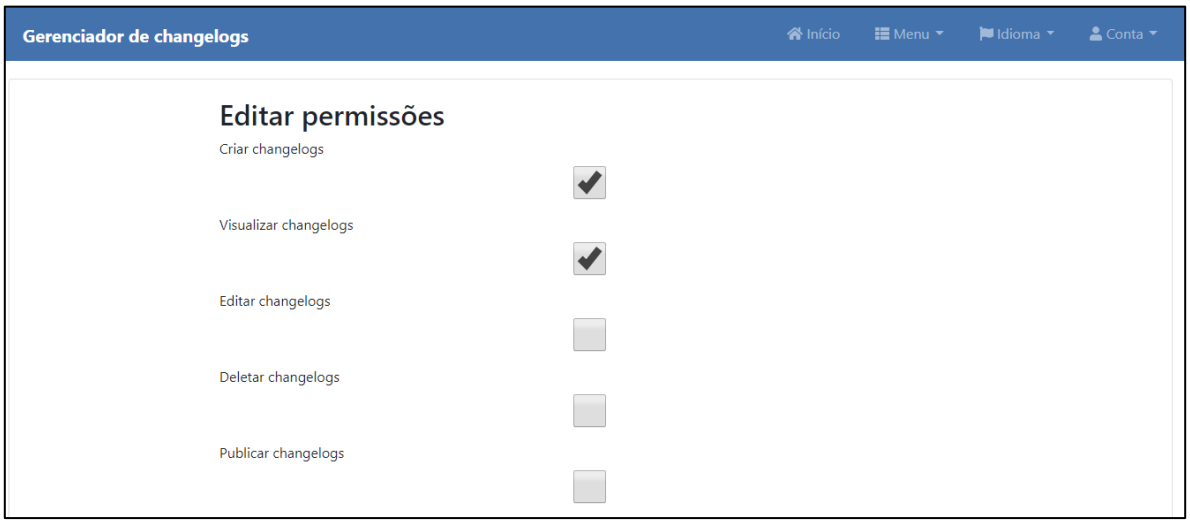
Figura 41 - Confirmar ação de aceitar ou recusar membro.



Fonte: Autor.

O gerenciamento de permissões é realizado na tela de membros, clicando no botão editar irá ser aberta a tela de gerenciamento de permissões que o membro pode executar, como na figura 42.

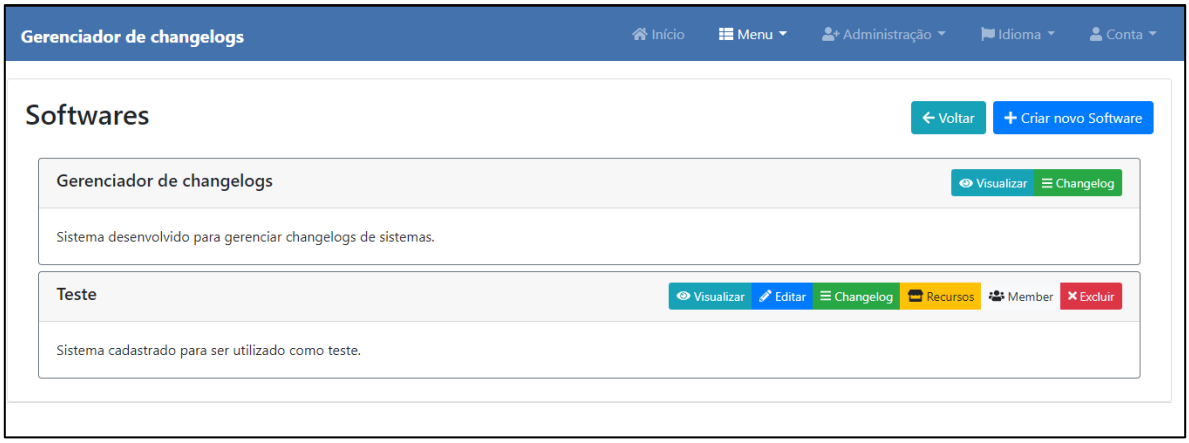
Figura 42 - Tela de gerenciamento de permissões.



Fonte: Autor.

Essas permissões são validadas em duas ocasiões, no *front-end* aonde elas são carregadas e os botões correspondentes às ações são exibidos ou escondidos, como na figura 43 que mostra a tela de *softwares*, como o membro que acabou de ser aceito só possui permissão para visualizar e publicar *changelogs* os demais botões não aparecem, em contraste a essa situação para o *software* Teste, cujo esse usuário é o criador, todos os botões são exibidos, pois o mesmo possui as devidas permissões.

Figura 43 - Validação de permissão no *front-end*.



Fonte: Autor.

Outra ocasião em que essas permissões são validadas são nas requisições, onde também é carregado o membro com suas permissões e é efetuada a devida validação para a ação, a próxima imagem mostra tal validação no *endpoint* em que é feita a requisição para a publicação de um *changelog*.

Figura 44 - Validação de permissão *back-end*.

```
public Boolean isMemberAllowedToPublishChangelogs(final Changelog changelog) {

        final Optional<String> userLogin = SecurityUtils.getCurrentUserLogin();

        if (userLogin.isPresent()) {
            final Optional<User> user = userRepository.findOneByLogin(userLogin.get());

            if (user.isPresent()) {
                final Member member = memberService.findMemberByChangelogSoftwareAndUserId(changelog, user.get());

                final Optional<Permission> permission = permissionRepository.findById(member.getPermission());

                if (permission.isPresent()) {
                    return permission.get().isCanPublishChangelog();
                }
            }
        }

        return false;
    }
```

Fonte: Autor.

No método apresentado na figura 44 é realizada uma busca pela permissão do membro, através do *software* que está no ligado ao *changelog* e o usuário que está acessando o sistema.

Assim encerrando o desenvolvimento do sistema proposto por este trabalho, todo o código fonte está disponível no repositório do *BitBucket*, criado para versionar o fonte deste

sistema. O *link* para acesso ao código-fonte é: https://bitbucket.org/makiasake/changelog-manager/src/master/.

4.4 EXECUTANDO A APLICAÇÃO

Para executar a aplicação desenvolvida neste trabalho é necessário seguir alguns passos, primeiramente será necessário um serviço *tomcat* rodando na maquina, o seu instalador e documentação podem ser encontrados no *link: https://tomcat.apache.org/download-80.cgi*. Qualquer versão maior ou igual a oito devem ser capazes de executar a aplicação desenvolvida. Outro serviço que deve estar sendo executado no computador que irá disponibilizar essa aplicação, é o do *MongoDB*, para instala-lo recomenda-se seguir as instruções de seu manual de instalação disponível no *link: https://docs.mongodb.com/manual/installation/*. Por ultimo deve-se instalar também o *Maven* disponível em: https://maven.apache.org/download.cgi. Este ultimo é utilizado para compilar o fonte do sistema que está disponível no link: https://bitbucket.org/makiasake/changelog-manager/src/master/.

Realizado o *download* do repositório, e com todos os serviços sendo executados na maquina e o *maven* instalado, é possível então compilar o pacote *war* acessando a pasta raiz do repositório e executando o comando: *mvn –Pprod clean package*, lembrando que a sintaxe do comando pode mudar dependendo do sistema operacional da maquina, em sistemas *Linux* seria necessário executar o comando como *./mvn –Pprod clean package*. Devido a essas particularidades dos sistemas operacionais e dos ambientes que a aplicação pode ser executada, é recomendada a leitura das documentações citadas.

CONCLUSÃO

Este trabalho apresentou *changelogs* de sistemas, expondo sua aplicabilidade no registro de alterações de sistemas para beneficiar usuários e desenvolvedores. O trabalho também apresenta o desenvolvimento de uma aplicação *web* para gerenciar e disponibilizar esses *changelogs* de sistema seguindo uma estrutura compreensiva baseada no modelo apresentado pelo projeto *Keep a Changelog* ou mantenha um *changelog*. A estrutura apresentada pelo mantenha um *changelog* aliada ao sistema desenvolvido nesse trabalho proporciona um meio simples de gerenciar e disponibilizar os *changelogs*. Podendo ser utilizado para a consulta das entradas por usuários do sistema, analistas, gerentes, desenvolvedores ou qualquer pessoa que precise saber sobre o histórico de um sistema.

O sistema faz isso através da estrutura desenvolvida para cadastrar *softwares* seus recursos, *changelogs*, entradas e membros, também fornecendo uma forma de publicar esse histórico de maneira incremental, possibilitado a adição das mudanças que ocorrerem no desenvolvimento dos sistemas dentro de um período cronológico e ao encerrar esse período de mudanças fornecendo um meio de publicar esse histórico para os usuários do sistema.

Apesar do conceito de *changelog*, como apresentado no *GNU Coding Standards* já providenciar uma estrutura para o registro deste histórico, a maneira como faz isto pode ser considerada insuficiente e atualmente até mesmo antiquada.

Utilizando a estrutura proposta pelo projeto mantenha um changelog foi então desenvolvida a aplicação que acompanha este trabalho, adicionando a essa estrutura a possibilidade do gerenciamento e distribuição dos *changelogs* de um sistema. Sendo que geralmente esse registro é realizado através de um arquivo junto com o código fonte do sistema.

Com isso então se espera que o presente trabalho contribua para aumentar a relevância do assunto e que o sistema desenvolvido neste trabalho possa ser utilizado por aqueles que desejem melhor gerenciar e disponibilizar *changelogs* de sistemas. Dito isso, é claro que a aplicação desenvolvida ainda pode ser melhorada, para o futuro desse sistema nota-se que as descrições das entradas poderiam permitir a inclusão de imagens ou vídeos, sendo que atualmente a mesma só permite texto. Outra funcionalidade que poderá ser implementada é

um modulo de conversas entre pessoas que contribuem nos *changelogs* de um mesmo sistema, e saindo desse principio também poderia existir um componente de comentários, para que usuários ou membros pudessem comentar sobre os *changelogs*.

Além de este trabalho discorrer sobre os *changelogs*, ao utilizar o *Framework Jhipster* para desenvolver o sistema, demonstra que a utilização de *frameworks* contribui para desenvolvedores num âmbito geral, facilitando o trabalho dos desenvolvedores por fornecer convenção sobre configuração, que possibilitou que o desenvolvimento da aplicação deste trabalho fluísse sem ter que se preocupar em configurar todo um ambiente de sistema. Outra questão é também a facilidade de se iniciar uma aplicação com as entidades principais seus serviços e repositórios (*CRUD* básico) já criados, possibilitando que o desenvolvedor invista seu foco no desenvolvimento das regras de negocio do sistema.

REFERÊNCIAS

LACAN, OLIVIER. ***Keep a Changelog***. 2017. Disponível em: <https://keepachangelog.com/en/1.0.0/>. Acesso em: 12 nov. 2018.

SOMMERVILLE, IAN. **Engenharia de Software: 9ª Edição.** 9ª.ed. São Paulo: Pearson Prentice Hall, 2013.

OBJECT MANAGEMENT GROUP. ***OMG® Unified Modeling Language® (OMG UML®): Version 2.5.1.*** **2017**. Disponível em: <https://www.omg.org/spec/UML>. Acesso em 14 nov. 2018.

STALLMAN, RICHARD. et al. **GNU CODING STANDARDS.** Free Software Foundation, Inc, 2018. Disponível em: <https://www.gnu.org/prep/standards/standards.pdf>. Acesso em: 14 nov. 2018.

KEETON, B.J. **Changelog and Release Notes Explained for Non-Developers**, 23 out. 2017. Disponível em: <https://www.elegantthemes.com/blog/resources/changelog-and-release-notes-explained-for-non-developers>. Acesso em: 13 nov. 2018.

APÊNDICE A

rc-yo.json


```
{
  "generator-jhipster": {
    "baseName": "changelog-manager",
    "packageName": "changelog_manager",
    "packageFolder": "changelog_manager",
    "authenticationType": "jwt",
    "hibernateCache": "no",
    "clusteredHttpSession": false,
    "websocket": false,
    "databaseType": "mongodb",
    "devDatabaseType": "mongodb",
    "prodDatabaseType": "mongodb",
    "searchEngine": false,
    "useCompass": false,
    "buildTool": "maven",
    "enableTranslation": true,
    "nativeLanguage": "pt-br",
    "applicationType": "monolith",
    "testFrameworks": [
      "gatling",
      "protractor"
    ],
    "languages": [
      "pt-br",
      "en"
    ],
    "serverPort": 8080,
    "jhipsterVersion": "5.3.0",
    "enableSocialSignIn": true,
    "useSass": false,
```

```
    "jhiPrefix": "jhi",
    "messageBroker": false,
    "serviceDiscoveryType": false,
    "clientPackageManager": "yarn",
    "clientFramework": "angularX",
    "jwtSecretKey":
"bXktc2VjcmV0LWtleS13aGljaC1zaG91bGQtYmUtY2hhbmdlZC1pbi1wcm9kdWN0aW9u
LWFuZC1iZS1iYXNlNjQtZW5jb2RlZAo=",
    "cacheProvider": "no",
    "enableHibernateCache": false,
    "reactive": false
  }
}
```

APÊNDICE B

changelog-jdl.jh

```
entity Changelog {
        title String,
        description String,
        published Boolean,
        software String
}

paginate Changelog with infinite-scroll
service * with serviceClass
```

entry-jdl.jh

```
entity Entry {
        title String,
        description String,
        status EntryStatus,
        type EntryType,
        resource String,
        changelog String
}

enum EntryStatus {
   REVIEWED, OPEN
}

enum EntryType {
        ADDED, CHANGED, DEPRECATED, REMOVED, FIXED, SECURITY
}
```

```
paginate Entry with infinite-scroll
service * with serviceClass
```

member-jdl.jh

```
entity Member {
        software String,
        permission String,
   role MemberRole
}

enum MemberRole {
   MANAGER, ADMINISTRATOR, DEVELOPER, COMMON
}

paginate Member with infinite-scroll
service * with serviceClass
```

permission-jdl.jh

```
entity Permission {
   canCreateChangelog Boolean,
   canReadChangelog Boolean,
   canUpdateChangelog Boolean,
   canDeleteChangelog Boolean,
   canPublishChangelog Boolean,
   canCreateEntry Boolean,
   canReadEntry Boolean,
   canUpdateEntry Boolean,
   canDeleteEntry Boolean,
   canUpdateSoftware Boolean,
   canDeleteSoftware Boolean,
   canAcceptMember Boolean,
```

```
    canRemoveMember Boolean,
    canCreateResource Boolean,
    canReadResource Boolean,
    canUpdateResource Boolean,
    canDeleteResource Boolean
}

paginate Permission with infinite-scroll
service * with serviceClass
```

resource-jdl.jh

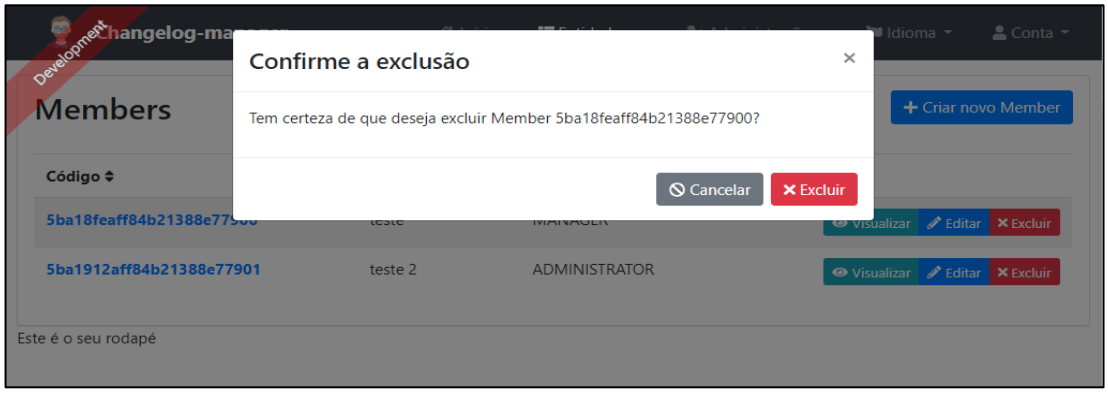
```
entity Resource {
        title String,
        description String,
        tagColor String,
        software String
}

paginate Resource with infinite-scroll
service * with serviceClass
```

software-jdl.jh

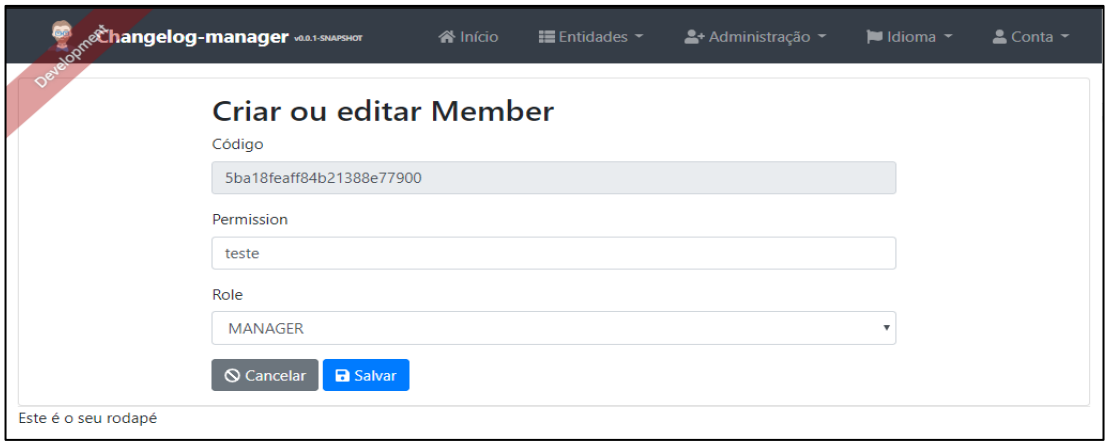
```
entity Software {
        title String,
    description String,
    publiclyVisible Boolean,
    acceptingNewMembers Boolean,
    imageUrl String
}
paginate Software with infinite-scroll
service * with serviceClass
```

APÊNDICE C

Tela gerada exclusão de membro.



Tela gerada edição de membro.



Tela gerada cadastro de membro.