UNIVERSIDADE FEDERAL DE SANTA CATARINA

Rede social para pesquisa e criação de projetos colaborativos

Yulle Pereira Ulguim

Florianópolis / Santa Catarina

2017/1

UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA
CURSO DE BACHARELADO EM SISTEMAS DE INFORMAÇÃO

Rede social para pesquisa e criação de projetos colaborativos

Yulle Pereira Ulguim

Trabalho de conclusão de curso apresentado
como parte dos requisitos para obtenção do grau
de Bacharel em Sistemas de Informação

Florianópolis / Santa Catarina

2017/1

Yulle Pereira Ulguim

Rede social para pesquisa e criação de projetos colaborativos

Trabalho de conclusão de curso apresentado como parte dos requisitos para obtenção do grau de Bacharel em Sistemas de Informação.

Orientador: Prof. Dr. Frank Augusto Siqueira

Banca Examinadora:

Prof. MSc. José Eduardo De Lucca

Prof. Dr. Leandro José Komosinski

RESUMO

A internet sempre manteve a cultura da colaboração e difusão do conhecimento. Seja com software livre ou com ambientes onde o conteúdo pode ser gerado por qualquer pessoa interessada em colaborar. A cada dia são lançadas milhares de aplicações nas lojas de aplicativos virtuais, grande parte por desenvolvedores independentes. A aplicação desenvolvida neste trabalho tem por objetivo ajudar a estes desenvolvedores e atuantes da área de tecnologia a encontrar colaboradores, sócios e outros profissionais interessados em participar na criação, implementação e execução de suas ideias, além de visar ampliar o alcance e a divulgação da visão colaborativa no mundo da tecnologia. A implementação do sistema foi realizada utilizando tecnologias atuais e padrões de desenvolvimento bastante utilizados no mercado de software atualmente.

Palavras-chave: Desenvolvimento colaborativo. Software. Open source. Rede social.

Lista de Figuras

Lista de Tabelas

SUMÁRIO

1. Introdução

Crowdsourcing, (em português, Colaboração coletiva), é a aglutinação das palavras *crowd* (grupo de pessoas) e *outsourcing* (terceirização). O termo foi cunhado em 2006 e adicionado no dicionário Merriam-Webster em 2011 com a seguinte definição:

*"*Crowdsourcing: Prática de obter serviços, ideias ou conteúdo através da contribuição de um grupo de pessoas, especialmente da comunidade virtual; ao invés de métodos tradicionais como empregados e fornecedores*"* .

Desde seu surgimento, a internet sempre manteve a cultura de compartilhamento da informação, seja ela de qualquer área. Fóruns, blogs, listas de discussão e tantos outros meios permitem a difusão do conhecimento. A própria comunidade *Open Source* sempre esteve muito à frente de outras organizações, tanto no compartilhamento do conhecimento tecnológico e filosófico que envolve a área de tecnologia da informação, quanto no compartilhamento de códigos-fonte dos mais variados tipos de software. Existem muitos tipos de licenças de distribuição de *software.* Entre as mais conhecidas e utilizadas estão a GNU *General Public License* (GPL)[1], a BSD License[2], e a *Apache License*[3]. Todas têm em comum o fato de incentivar o compartilhamento dos códigos-fonte e do conhecimento gerado pela aplicação ou produto criado.

Muitos dos projetos desenvolvidos de maneira coletiva, ou para serem utilizados de maneira coletiva, nasceram através da necessidade de eliminar um problema ou aperfeiçoar tarefas e situações na vida de uma comunidade, seja ela virtual ou real. Um exemplo de sistema colaborativo é a *Wikipedia*[4], uma enciclopédia virtual com mais 5 milhões de artigos em sua versão em inglês e quase 1 milhão na versão em português. Os usuários têm livre acesso para ler, editar e adicionar novos artigos; o que faz a

[1] http://www.gnu.org/licenses/gpl.html
[2] http://www.linfo.org/bsdlicense.html
[3] http://www.apache.org/licenses/LICENSE-2.0
[4] http://wikipedia.org

plataforma possuir 60 vezes mais palavras do que a segunda maior enciclopédia de língua inglesa, a Encyclopædia Britannica[5].

O crescimento da cultura de *startups* e do *home office* nos últimos anos, e o aumento da quantidade de dispositivos móveis no Brasil e no mundo, ocasionaram um forte crescimento no mercado de *software*. Hoje qualquer pessoa com um computador e acesso à internet pode aprender a programar e em pouco tempo lançar seu próprio aplicativo nas lojas virtuais. Segundo a Apple, em uma reportagem para o *International Business Time* em 2015, são submetidos mais de 1000 novos aplicativos móveis todos os dias na *AppStore*.

Apesar de todas as facilidades encontradas nos dias atuais para quem deseja ingressar no mercado de software de maneira independente, existe um problema que afeta muitos desenvolvedores e entusiastas autônomos que estão iniciando a carreira. Em conversas informais sobre o assunto, muitos desses profissionais apontaram para uma dificuldade vital no processo de criação de software: a falta de *networking* com pessoas de outras áreas que não a de programação. Em várias situações no desenvolvimento de uma aplicação, seja ela um jogo, um gerenciador de tarefas, um *player* de músicas, ou qualquer outra finalidade, precisamos de pessoas que não estão envolvidas diretamente em nossa área de atuação. Um designer para compor a identidade visual de nosso projeto ou desenhar personagens de um jogo, um engenheiro de som para ajudar a captar e criar sons de notificação, ou até mesmo um tradutor para auxiliar no processo de tradução do aplicativo. Encontramos bastante dificuldade em localizar pessoas de fora da nossa realidade e com interesses em comum para colaborar com a execução do nosso projeto.

Para tentar diminuir este problema, surgiu a ideia de desenvolver uma aplicação web na qual usuários possam se cadastrar, manter um perfil, criar e pesquisar projetos ou pessoas com interesses em comum para execução de projetos, fomentando assim, a indústria de software independente e a cultura do *crowdsourcing*.

[5] http://britannica.com

1.1 Objetivos

1.1.1 Objetivos gerais

Este trabalho tem como objetivo o desenvolvimento de um sistema que ajude pessoas com interesses em comum a interagirem para criação, desenvolvimento e manutenção de projetos colaborativos, fomentando a cultura de colaboração e a implementação de novas ideias no mercado de software.

1.1.2 Objetivos específicos

- Levantamento e estudo das tecnologias que poderão ser utilizadas para a implementação da aplicação web para anúncios e pesquisa de parceiros para o desenvolvimento de projetos colaborativos.
- Desenvolvimento de uma aplicação web que permita que usuários cadastrem um perfil pessoal com interesses, links para portifólio e redes sociais, e informações de contato; criem páginas de projetos que estão desenvolvendo e necessitam de outras pessoas, procurem por outros usuários já cadastrados que estão interessados em participar de projetos colaborativos e se candidatem para projetos.
- Publicação da aplicação desenvolvida em um servidor dedicado para validação e divulgação da mesma.

1.2 Metodologia

Este trabalho procura apresentar a possibilidade de reunir pessoas com interesses em comum para desenvolver e fomentar a cultura da colaboração dentro do processo de desenvolvimento de software e da área de tecnologia. O modelo de implementação adotado será baseado nos conceitos abordados no capítulo de fundamentos tecnológicos

e irá seguir os princípios da metodologia de desenvolvimento ágil de software e o Manifesto Ágil[6]. Ao final da implementação da aplicação, serão realizados testes de validação da interface com alguns usuários.

1.3 Organização do texto

Deste ponto em diante, o trabalho se encontra dividido nos seguintes capítulos:

- Fundamentos tecnológicos: descrição das tecnologias que serão utilizadas no desenvolvimento da aplicação.
- Trabalhos relacionados: mostra as principais aplicações disponíveis no mercado, seus pontos fortes e falhas.
- Projeto: descreve os requisitos funcionais e não-funcionais da aplicação, seu comportamento e funcionamento, *wireframes,* modelo de banco de dados e alguns diagramas UML.
- Desenvolvimento: descreve em detalhes o processo de implementação do sistema, levando em consideração os conceitos exibidos nos capítulos anteriores.

[6] http://agilemanifesto.org

2. Fundamentos Tecnológicos

Este capítulo apresenta as tecnologias utilizadas na implementação do sistema, que foram selecionadas com base no conhecimento adquirido durante o curso e, ao mesmo tempo, com o objetivo de obter um aprofundamento em algumas delas e conhecer as soluções que estão sendo utilizadas atualmente no mercado de software. O back-end é responsável por toda a lógica de negócio, tratamento de requisições e persistência e recuperação dos dados da aplicação, sendo inteiramente executado no lado do servidor. O front-end gerencia o comportamento das telas e toda a interação com o usuário e tem a lógica executada no navegador do usuário.

2.1 Back-end

A aplicação irá necessitar de um servidor que terá como responsabilidade o tratamento de todas as requisições feitas através da interface do usuário (front-end).

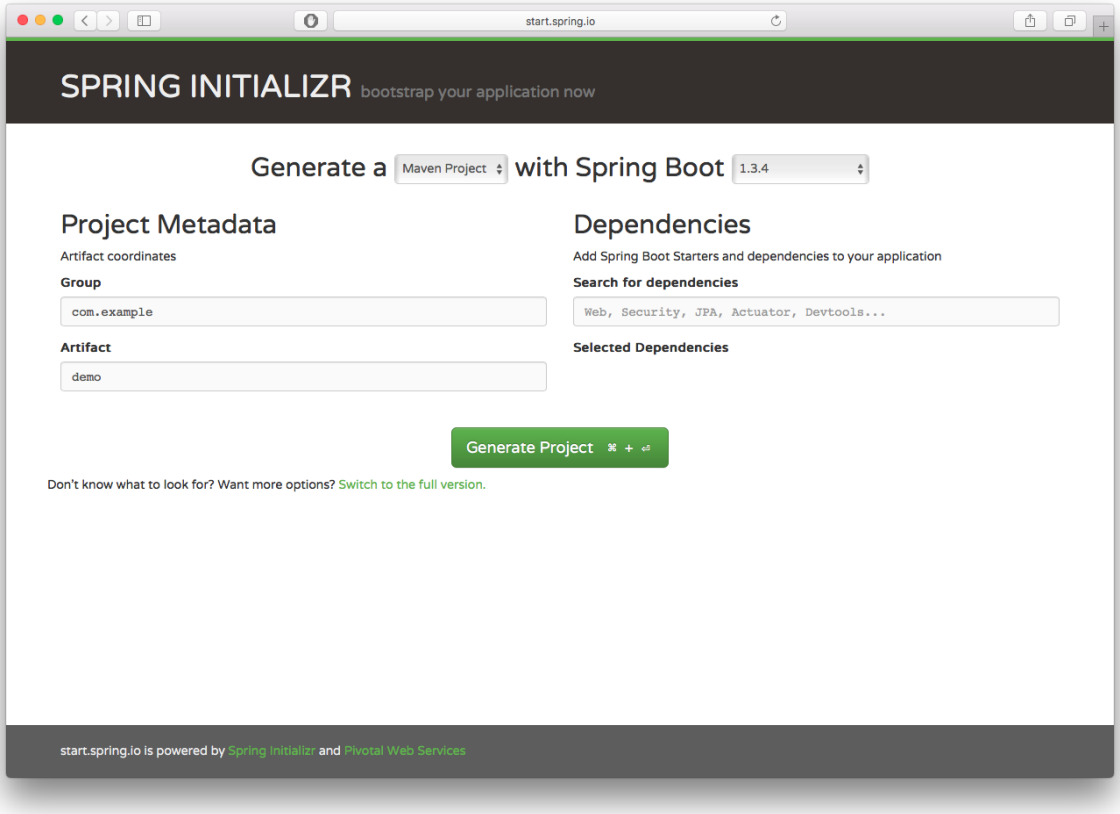
A persistência de dados será realizada utilizando PostgreSQL[7] e o acesso a esta camada, gerenciado pelo Hibernate[8]. A linguagem que será utilizada para desenvolver este lado do software é o Java[9], juntamente com o framework Spring Boot[10].

O Spring Boot é desenvolvido pela Pivotal Software[11]. Ele é baseado na arquitetura MVC (*model-view-controller*), um padrão amplamente difundido e utilizado em aplicações *web* e que define bem as responsabilidades de cada camada do *software*. O framework é baseado no já conhecido e bastante difundido Spring, e tem como objetivo facilitar o desenvolvimento e a publicação de aplicações *web*. Sua filosofia é a "*convention over configuration"*, ou seja, o programador não deve se preocupar com a configurações complexas do *framework*; a intenção é disponibilizar diversos pacotes pré-configurados

[7] http://postgresql.org
[8] http://hibernate.org
[9] http://java.com
[10] http://spring.io
[11] http://pivotal.io

como módulos, de modo que o desenvolvedor vai embutindo no projeto apenas as funcionalidades que deseja utilizar. A empresa disponibiliza um site, exibido na figura 1, onde o usuário pode pré-configurar uma aplicação *spring* com os módulos que deseja e baixá-lo para ser importado em sua IDE de desenvolvimento.

Figura 1 - Spring Initializr

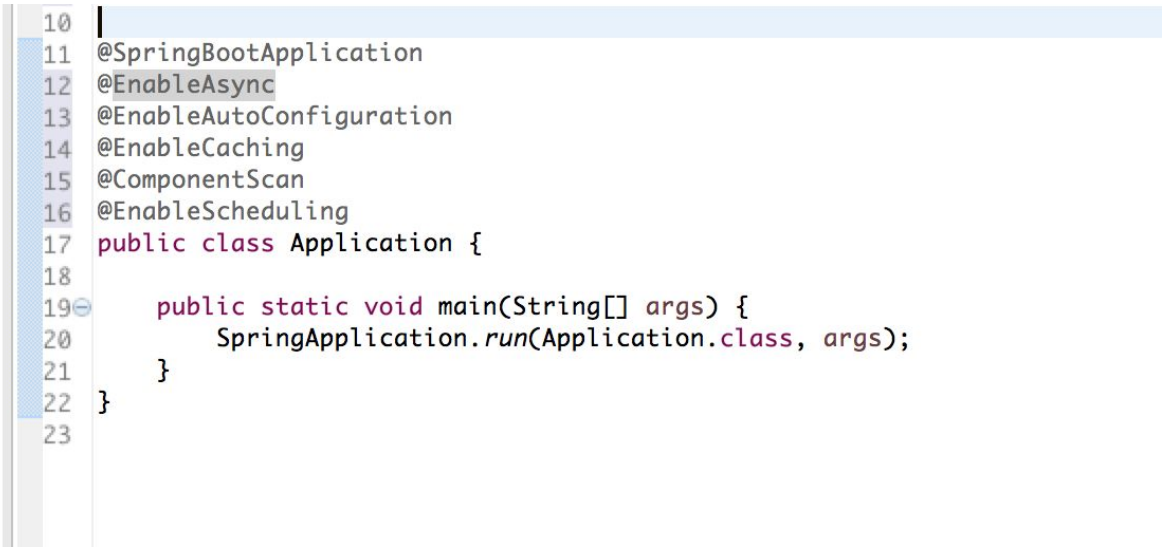


Fonte: http://start.pring.io

Algumas das características do Spring Boot:

- Configuração baseada em anotações: até antes da versão 4, o Spring era conhecido por ser um *framework* difícil de ser configurado, com praticamente todas as configurações sendo feitas através de XMLs numerosos e confusos. Na versão 4, lançada em dezembro de 2013, a configuração é feita principalmente por anotações nas classes e métodos do projeto, tornando mais prático e ágil o processo de desenvolvimento de aplicações. A figura 2 mostra alguns exemplos de *annotations.*

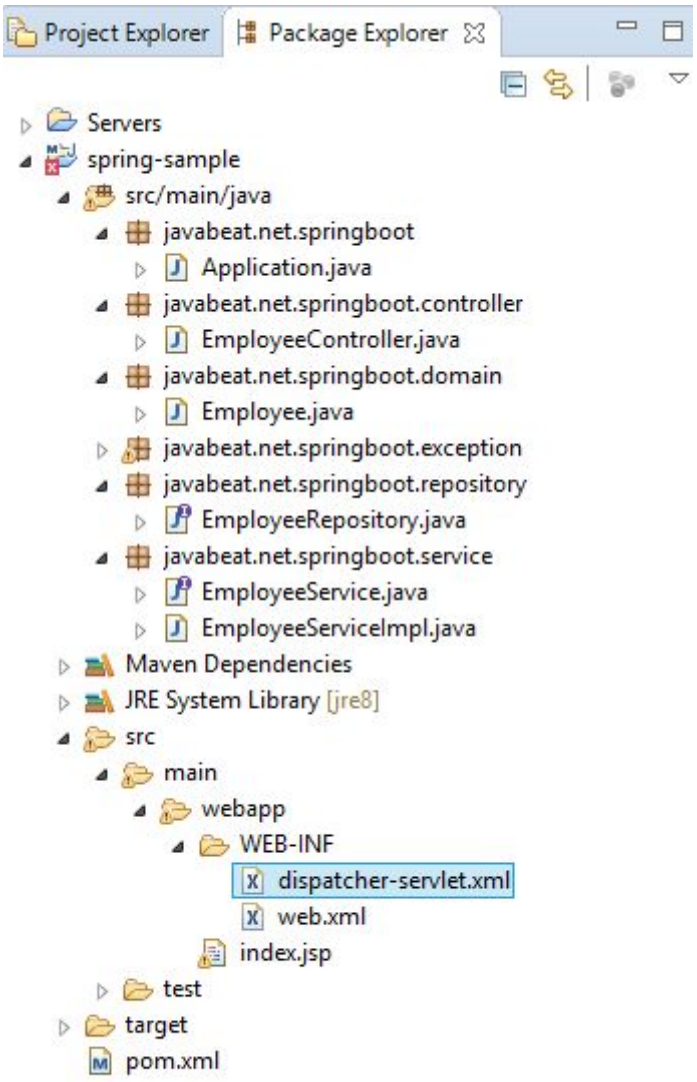
Figura 2 - Exemplo de annotations



Fonte: elaborada pelo autor

- Não depende de servidores de aplicação externos: normalmente quando desenvolvemos uma aplicação *web*, necessitamos de um servidor (Tomcat, Wildfly, Jetty, etc.) para publicar e gerenciar as instâncias do programa. A solução encontrada pelo Spring Boot para eliminar a necessidade de configurações externas é a utilização de *containers* de servidores embutidos em sua estrutura, como o Tomcat e o Jetty, que funcionam "*out-of-the-box*"; ou seja, não necessitam de configuração.
- Adaptável e escalável: dependendo da necessidade e do número de acessos no sistema, se torna necessário o *deploy* de múltiplas instâncias da mesma, um processo que pode ser demorado e perigoso, pois na grande maioria dos servidores disponíveis no mercado é preciso parar a execução das aplicações para configuração de novas instâncias. No *framework* da Pivotal nada disso é necessário. Como mostrado no tópico anterior, ele não depende de servidores de aplicação, podemos subir novas instâncias de nosso sistema e tudo funciona transparentemente, tanto para os desenvolvedores, quanto para os usuários. É possível utilizar diferentes servidores embarcados para cada instância e distribuir a carga sobre cada uma delas através de servidores *web*, como o Apache ou Nginx.

Figura 3 - Organização de uma aplicação Spring Boot



Fonte: elaborada pelo autor

A figura 3 mostra como os pacotes são organizados em uma aplicação Spring Boot. O pacote raiz, no exemplo "javabeat.net.springboot", possui o método *main*, que inicia a aplicação. O *framework* faz o escaneamento automático de todos os serviços, repositórios e *managers* que estão abaixo do pacote principal. O repositório "EmployeeRepository", é a interface utilizada para recuperar, inserir e atualizar dados referentes ao objeto Employee. Assim que a aplicação é executada, esta classe estará disponível para uso em todas as outras classes que estão em pacotes abaixo de "javabeat.net.springboot".

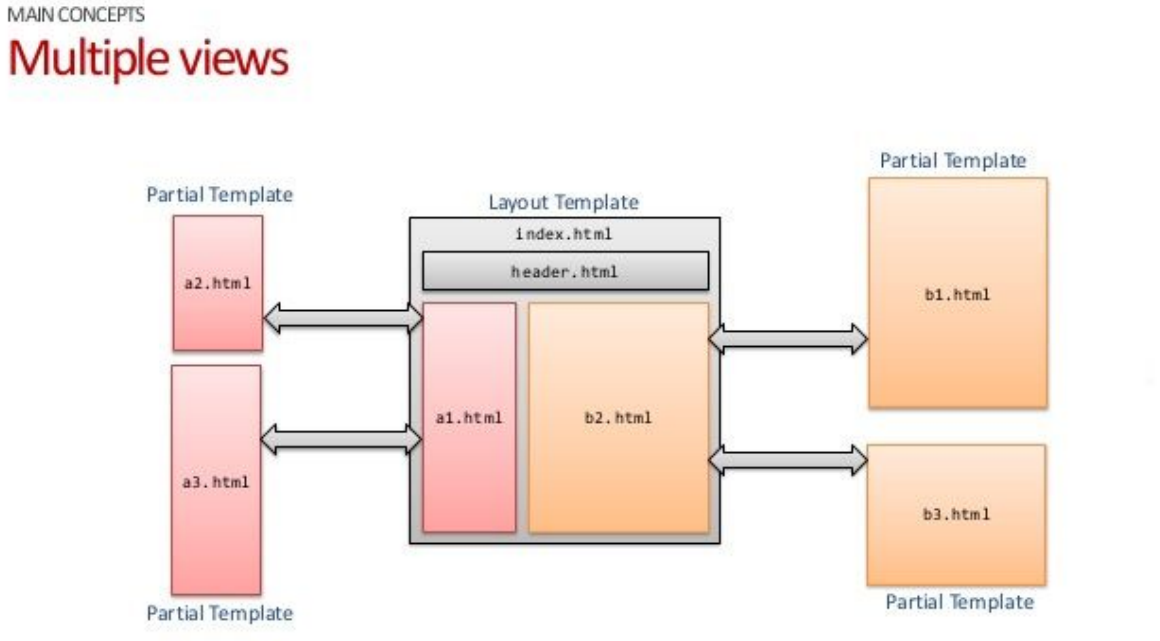
2.2 Front-end

O lado do cliente será implementado utilizando HTML e AngularJS.

O AngularJS é um framework *front-end* para aplicações web escrito em javascript e desenvolvido e mantido pela Google. Segundo o site Libscore[12], que coleta dados de uso dos principais *frameworks* do mercado, seu uso cresce a cada dia entre os desenvolvedores web. Possui características MVC, *single-page application* e *two-way data binding*, que facilitam e aceleram o processo de desenvolvimento do front-end.

Single-page application é o ato de disponibilizar toda a aplicação em uma única página, alterando somente o necessário a cada requisição do usuário. O Angular utiliza uma tecnologia chamada *partials* para isto, que consiste em modificar apenas a *div* definida como tal, economizando download e upload de dados e tornando as chamadas mais eficientes.

Figura 4 - Funcionamento das templates no AngularJS.



Fonte: http://image.slidesharecdn.com/95/working-with-angularjs-42-638.jpg

[12] http://libscore.com/#angular

Como podemos ver na figura 4, o *layout* da página, formado pelos arquivos index.html e header.html, permanece sem alterações; o que muda efetivamente são as *templates* parciais, de acordo com as funcionalidades da aplicação que o usuário invoca.

Para exemplificar, vamos imaginar uma aplicação que gerencia uma escola. Entre as principais funcionalidades temos o cadastro de alunos, professores e salas de aula com seus respectivos horários e matérias. O menu e nome da pessoa logada no sistema ficaria disposto no arquivo index.html, pois esta informação nunca vai ser alterada durante a sessão do usuário atual. Quando o usuário acessar outra tela a partir do menu, o Angular vai automaticamente procurar pelo arquivo correspondente a esta função e carregá-lo na área definida pelo desenvolvedor como *partial*.

O que torna o AngularJS prático para o desenvolvimento de aplicações *web* diante de outros frameworks é um recurso muito interessante chamado *two-way data binding.* Nesse sistema, uma variável criada no *controller* da página é vinculada diretamente à tela do usuário e vice-versa. Para evitar conflito entre nomes de variáveis, cada uma pertence ao seu próprio escopo e não pode ser chamada de outro controlador.

O *framework* permite também a criação de diretivas, que podem ser chamadas diretamente da tela na forma de *tags* HTML. O módulo principal do Angular já possui algumas diretivas para resolver problemas triviais, como condicionais (*ng-if*) e *loops* (*ng-repeat*). Estas expressões são normalmente utilizadas para exibir dinamicamente variáveis nos *templates* HTML.

O exemplo mostrado na figura 5 ilustra a utilização de diretivas, templates e variáveis de escopo.

O módulo "*myApp"* define o *controller* "*MyController"*, que possui uma lista com três livros, que será utilizada para montar uma lista ordenada na tela do usuário. Este *template* usa diretivas e expressões que já vem por padrão no Angular (*ng-if, ng-repeat, orderBy)*.

Figura 5 - Exemplo de utilização de uma diretiva no AngularJS

```
<html ng-app="myApp">
 <head>
  <script src="angular.min.js"></script>
  <script>
    angular.module('myApp', []);
    function MyController($scope) {
      $scope.books = [
        { title: "One Thousand and One Nights" },
        { title: "Where the Wild Things Are", author: "Sendak" },
        { title: "The Hobbit", author: "Tolkien" },
      ];
    }
  </script>
 </head>
 <body ng-controller="MyController">
  <ul ng-repeat="b in books | orderBy:'title'">
   <li>
    <i>{{b.title}}</i>
    <span ng-if="b.author">by {{b.author}}</span>
   </li>
  </ul>
 </body>
</html>
```

Fonte: http://journal.code4lib.org/articles/10023

3. Trabalhos Relacionados

Para o desenvolvimento deste trabalho, foram pesquisadas diversas aplicações com funcionalidades que se assimilam ao sistema desenvolvido.

Existem diversas aplicações disponíveis no mercado que possuem objetivos parecidos com o sistema desenvolvido neste trabalho. Alguns com múltiplas funcionalidades, como aplicativos para desenvolvimento de um projeto de software desde seu início; outros com funcionalidades únicas e bem estabelecidas, como repositórios de código.

Abaixo, são listadas algumas delas com suas características e particularidades.

http://www.guru.com/ - Contratar *freelancers* online. Diversidade de áreas de atuação (desenvolvimento, design, finanças). Todos os trabalhos são remunerados. Tem funcionalidades similares às desenvolvidas neste projeto: possibilidade de criar um perfil pessoal com suas experiências profissional e possibilidade de criar uma página para o projeto que será desenvolvido.

https://www.freelancer.com/ - Voltado para o mercado de software e web. Tem mais de 22 milhões de profissionais e empresas registrados. Assim como no sistema desenvolvido neste trabalho, os usuários se candidatam aos projetos e dependem da aceitação do dono do mesmo. Todos os trabalhos são remunerados.

https://www.toptal.com - Foco no mercado de software e design. Todos os trabalhos são remunerados. Ao cadastrar um projeto, o sistema faz uma seleção dos candidatos que possuem o perfil que mais se encaixa no trabalho. A aplicação desenvolvida possui uma funcionalidade parecida: na tela principal são mostrados projetos de acordo com o perfil e as habilidades do usuário.

https://www.99freelas.com.br/ - Voltado para o mercado brasileiro. Bastante utilizado e conhecido no Brasil por desenvolvedores de software e designers. Nesta plataforma, os profissionais se cadastram e os contratantes pesquisam e negociam com os usuários. Todos os trabalhos são remunerados.

http://www.prolancer.com.br/ - Voltado para o mercado brasileiro. Permite que empresas divulguem vagas de emprego. Os trabalhos são remunerados. Adquirido recentemente pelo freelancer.com, terá seus usuários migrados para a plataforma australiana.

http://sourceforge.net/ - Fundado em 1999, foi um dos primeiros a oferecer gratuitamente a hospedagem de projetos *open source*. Possui diversas funcionalidades, dentre elas: *wiki* para o projeto, *reviews* dos projetos, listas de emails e mural de notícias e atualizações. Esta última, foi implementada no sistema desenvolvido neste trabalho. Os usuários que participam do projeto podem criar postagens com notícias e discussões na página do projeto.

4. Projeto

4.1 Visão geral

O sistema proposto será implementado buscando resolver os problemas e situações abordados no capítulo de introdução deste trabalho. O objetivo final é a criação de uma aplicação web que permita que pessoas interessadas em utilizar o sistema possam se cadastrar no mesmo. Os usuários cadastrados poderão manter um perfil pessoal com foto, interesses, experiências em projetos e trabalhos passados, links para portifólio e redes sociais de sites externos, além de endereços de email e telefones de contato, Skype e outras ferramentas de comunicação. Além disso, os usuários poderão também criar e pesquisar páginas sobre projetos que estão sendo desenvolvidos por outras pessoas ou que precisam de voluntários para serem inicializados. Por último, a aplicação contará com uma tela que exibirá projetos relevantes de acordo com o perfil do usuário que está acessando o sistema.

Após implementado, o sistema irá ser hospedado em um servidor dedicado na nuvem. Este servidor irá possuir um servidor de banco de dados PostgreSQL para persistência de dados da aplicação e um servidor *web* Apache para gerenciar as conexões HTTP, permitindo assim a possibilidade de executar múltiplas instâncias da aplicação, balanceando a carga entre todas elas.

4.2 Requisitos funcionais e não-funcionais

4.2.1 Prioridade dos requisitos

Para tornar mais eficiente o desenvolvimento do sistema, foram adotadas prioridades para cada requisito do projeto baseadas no objetivo final do projeto e em conversas informais com potenciais usuários da aplicação:

- **Indispensável:** essencial para o funcionamento da aplicação. Sem a implementação destes requisitos a publicação e utilização do sistema não é possível.
- **Importante:** requisito que mesmo que não esteja implementado, permite o funcionamento da aplicação, mesmo que de forma não satisfatória. Estes requisitos devem ser implementados, mas caso não o sejam, a aplicação poderá ser utilizada mesmo assim.
- **Desejável:** não compromete a utilização geral do sistema. A aplicação funcionará de forma satisfatória se algum requisito com esta prioridade não for implementado. Eles serão implementados após os de prioridade mais elevada.

4.2.2 Requisitos Funcionais

A tabela 1 apresenta os requisitos funcionais do sistema e suas respectivas prioridades.

Tabela 1 - Requisitos funcionais

Identificador	Nome	Descrição	Prioridade
[RF01]	Cadastrar usuário	Permitir o cadastro de qualquer pessoa interessada em utilizar o sistema	Indispensável
[RF02]	Editar perfil do usuário	Permitir que o usuário edite seu perfil pessoal	Indispensável

[RF03]	Criar projeto	Permitir que o usuário crie um projeto	Indispensável
[RF04]	Editar projeto	Permite que o usuário edite um projeto que publicou	Importante
[RF05]	Pesquisar projeto	Permite que o usuário faça pesquisa de projetos publicados	Importante
[RF06]	Pesquisar projeto baseado na localidade	Permite que o usuário faça pesquisa de projetos publicados perto de sua localização atual	Desejável

4.2.3 Requisitos Não-Funcionais

A tabela 2 apresenta os requisitos não-funcionais que serão considerados durante o desenvolvimento do sistema.

Tabela 2 - Requisitos não-funcionais

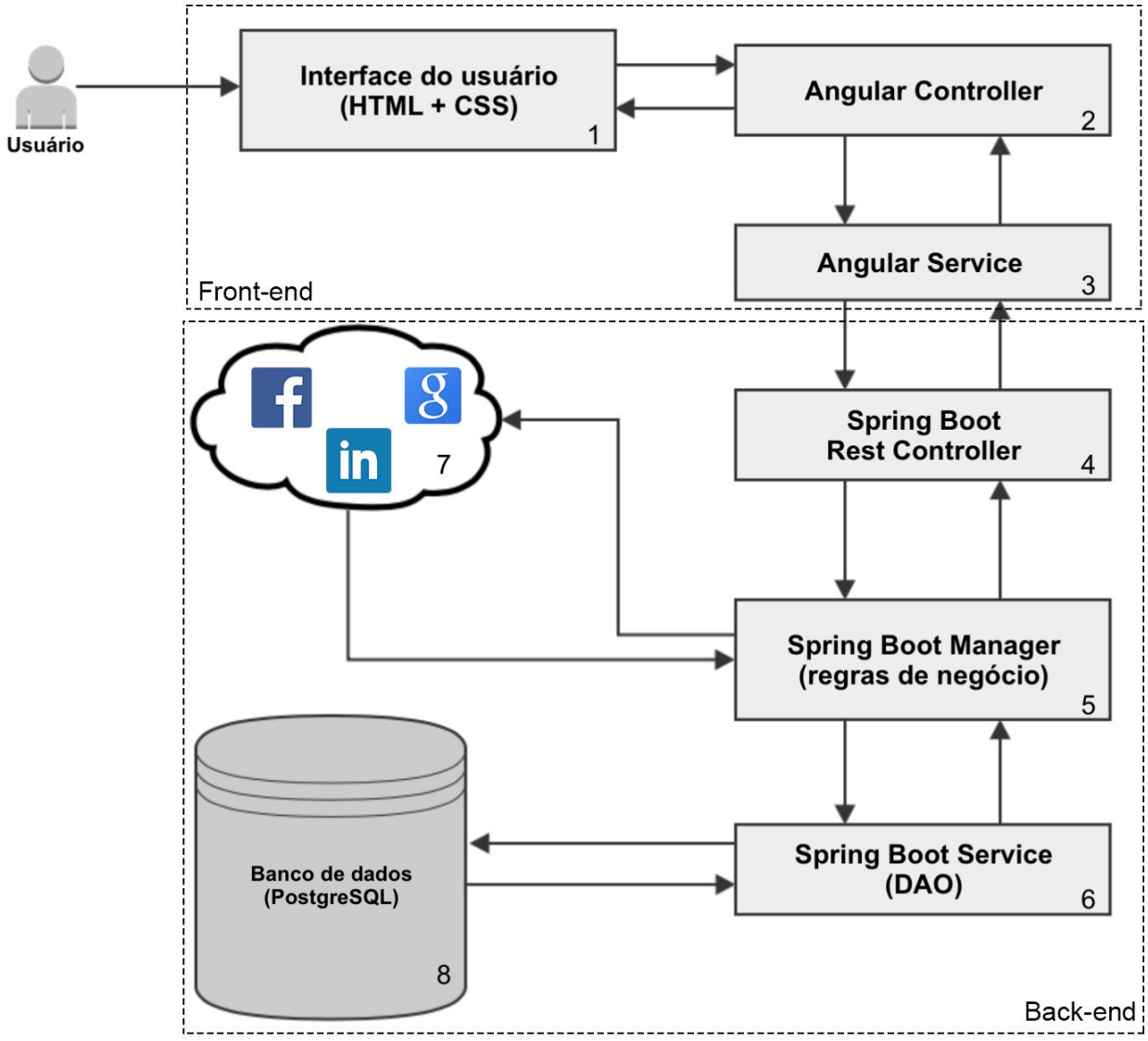
Identificador	Nome	Descrição	Prioridade
[RNF01]	Aplicação web	O sistema será implementado para rodar em um navegador web	Indispensável
[RNF02]	Arquitetura REST	O sistema será implementado utilizando os padrões REST de desenvolvimento	
[RNF03]	Login via APIs	O sistema irá permitir login simplificado através de APIs (Google, Facebook, LinkedIn)	Importante

[RNF04]	Divisão de arquitetura do sistema através de camadas	O sistema irá utilizar o padrão MVC como arquitetura de desenvolvimento	Indispensável
[RNF05]	Importação de informações do usuário através da API do LinkedIn	O sistema permitirá a importação das informações do perfil do LinkedIn do usuário	Importante
[RNF06]	Interface responsiva	O sistema terá interface responsiva e será suportado por navegadores de dispositivos móveis	Desejável

4.3 Arquitetura

A aplicação será implementada utilizando o modelo MVC (*model-view-controller*), uma arquitetura bem definida e organizada, onde cada componente tem suas funções determinadas. Isto garante maior facilidade na manutenção e reusabilidade do código. Se em algum momento da vida do software houver necessidade de alteração, por exemplo, da interface e do *framework* que a gerencia, isso pode ser feito com muito mais eficiência, visto que será preciso reprogramar apenas a parte do sistema responsável por gerenciar estas funções. O sistema será desenvolvido utilizando a arquitetura REST. A figura 6 mostra a arquitetura geral do sistema, que é detalhada abaixo.

Figura 6 - Arquitetura geral do sistema



Fonte: elaborada pelo autor

Front-end

1. **Interface do usuário**: As páginas do sistema. É responsável por apresentar a interface ao usuário. Utiliza tecnologias como HTML, CSS e JavaScript.
2. **Angular Controller**: É responsável pelo comportamento da interface. Ações de clique, carregamento de dados e manipulação das telas são realizadas nesta camada.

3. **Angular Service**: Pode ser considerado o DAO do front-end. Tem como responsabilidade enviar os dados para o back-end e recuperar informações já registradas no banco de dados.

Back-end

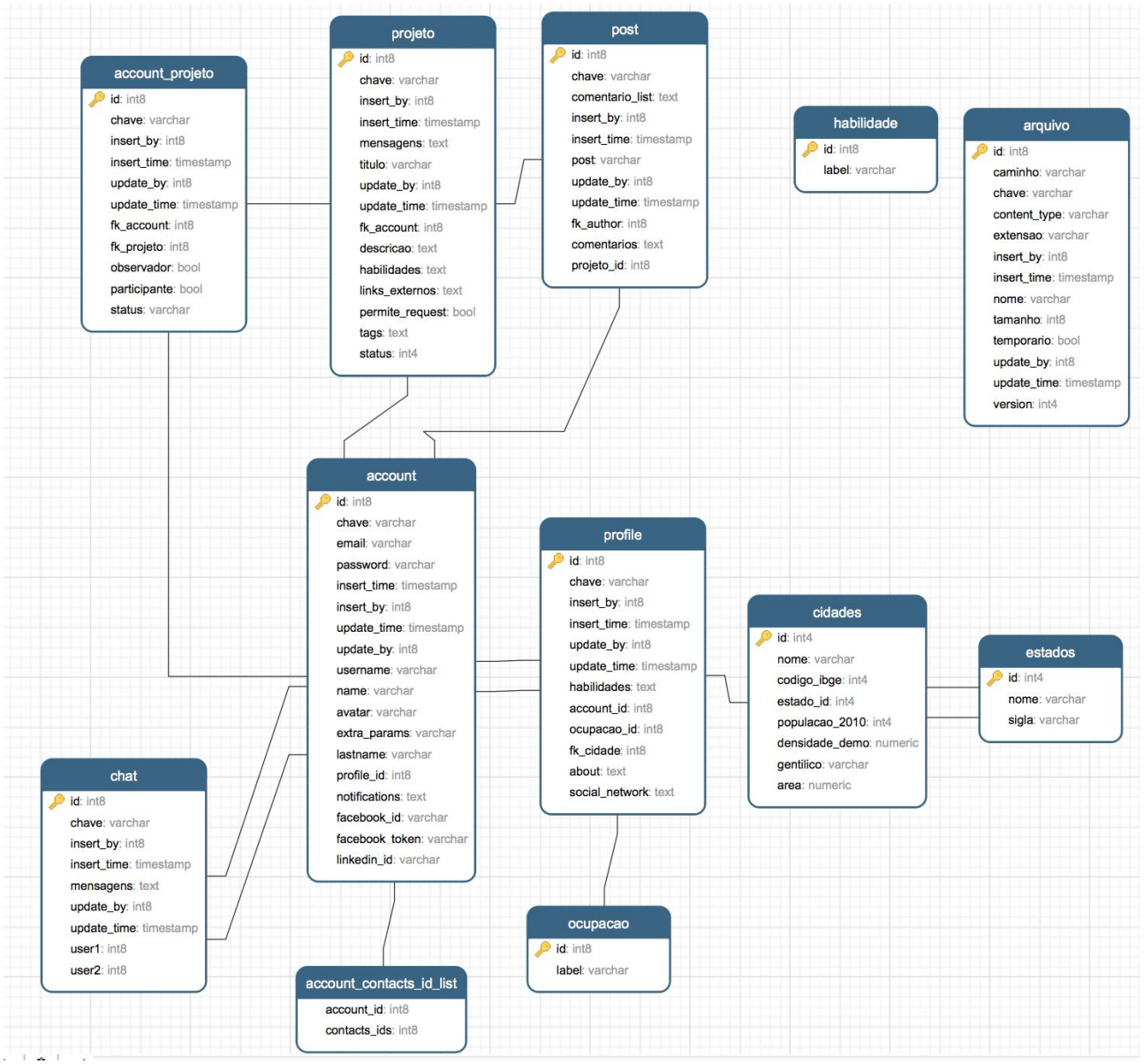
4. **Spring Boot Controller**: Recebe as requisições do front-end, e as encaminha para os *managers*. Esta camada não possui regras de negócio.
5. **Spring Boot Manager**: Aqui é realizado todo o processamento dos dados. Validações, filtros, tratamento de exceções e todas as regras de negócio são feitas nesta camada da aplicação.
6. **Spring Boot Service**: Responsável pela persistência e recuperação dos dados dos usuários. Nessa camada será utilizado um *framework* de mapeamento objeto-relacional.
7. **APIs de terceiros**: APIs de autenticação e recuperação de informações dos usuários em sistemas de terceiros.
8. **Banco de dados**: banco de dados relacional onde as informações e configurações dos usuários são persistidas e manipuladas.

4.4 Banco de dados

Utilizando como base as funcionalidades descritas nos requisitos do projeto e a arquitetura mostrada na seção anterior, foi elaborado o modelo do banco de dados.

O banco de dados tem como papel principal armazenar os dados da aplicação e garantir a integridade de dados do sistema, por meio da verificação de identificadores de usuários e senhas, armazenar todos os projetos publicados, perfis de usuários, notificações e todas as informações necessárias para o funcionamento ideal do software. O modelo foi desenvolvido com o objetivo de ser facilmente adaptável conforme evolução e necessidade do sistema em suas iterações futuras. Na figura 7 temos o resultado da modelagem do banco de dados.

Figura 7 - Modelo do banco de dados



Fonte: elaborada pelo autor

5. Desenvolvimento

5.1 Back-end

É indispensável a implementação de um servidor que irá conter a lógica de negócios, que será responsável por tratar as requisições do lado do cliente. Para isso, utilizaremos a linguagem de programação Java em conjunto com o framework Spring Boot, e o SGBD (Sistema Gerenciador de Banco de Dados) PostgreSQL.

O Spring Boot possui servidores HTTP embarcados e que não requer configuração alguma de arquivos XML para funcionar. Qualquer customização pode ser realizada via programação. Spring Boot facilita bastante a criação de servidores REST, encapsulando várias questões e rotinas que poderiam levar dias para serem configuradas e implementadas do início. No caso da aplicação que está sendo desenvolvida neste projeto, o Spring Boot gerencia de forma transparente o tratamento de requisições REST através de seus controladores, a injeção e controle de dependência de todos os componentes do programa. Utiliza a filosofia "convenção sobre configuração", e a maioria das configurações básicas são abstraídas por anotações e padronização de nomes de classes e métodos, o que facilita e agiliza muito a construção do programa, fazendo com que o desenvolvedor tenha como foco a resolução do problema acima de qualquer outra coisa. A ferramenta pode ser utilizada em pequenos servidores e até em aplicações com alto número de requisições.

Conforme definido no capítulo 4, seção 4.3, a camada de acesso aos serviços de gerenciamento da nossa aplicação será uma API REST, disponibilizada através de *controllers* REST, que simplificadamente são classes que expõem ações e informações ao cliente, no caso a camada *front-end*, acessada pelos usuários da plataforma. Todas as respostas são serializadas e entregues no formato JSON, padrão de comunicação amplamente utilizado em aplicações web modernas.

O back-end possui uma organização e hierarquia de arquivos e pacotes bem definida para facilitar o desenvolvimento e manutenção do código escrito. O nome define explicitamente o que está contido em cada pacote. O pacote *controller* armazena todos os

controladores de nossa aplicação, o pacote *manager*, todos os *managers* e no *service* estão todas as classes que tem como objetivo a comunicação com o banco de dados. Isso se repete em toda a aplicação. Cada classe também possui um nome que identifica exatamente o papel que ela tem no programa. Por exemplo, a classe ChatManager tem toda a regra de negócio para gerenciar os *chats* dos usuários. As entidades estão todas contidas no pacote *entity*, e os *DTO's*, que são as entidades utilizadas para expor os dados na tela, no pacote *view*.

Os *controllers* são a primeira camada do servidor, e todas as requisições são recebidas por eles, isolando completamente as regras de negócio do ambiente externo.

Figura 8 - ProjetoController

```
@RestController
@ControllerSecurity(ControllerSecurity.Security.PRIVATE)
@RequestMapping(value="/projeto", produces = MediaType.APPLICATION_JSON_UTF8_VALUE)
public class ProjetoController extends TCCBaseController {
  @Autowired private ProjetoManager projetoManager;

  @RequestMapping(method = RequestMethod.GET)
  public MeusProjetosDTO listMyProjects() throws ValidationException {
     MeusProjetosDTO dto = new MeusProjetosDTO();
     dto.meusProjetos = projetoManager.list(getProfile());
     dto.projetos = projetoManager.listQueEuParticipo(getProfile());
     return dto;
  }

  @RequestMapping(value="/{key}", method = RequestMethod.GET)
  public ProjetoDTO load(@PathVariable("key") String key) throws ValidationException {
     ProjetoView view = new ProjetoView();
     view.setKey(key);
     ProjetoDTO dto = new ProjetoDTO();
     dto.projeto = projetoManager.load(getProfile(), view);
     if (dto.projeto.getMeuProjeto() || dto.projeto.getSouParticipante()) {
        dto.posts = projetoManager.loadPosts(getProfile(), view);
        dto.participantes = projetoManager.loadParticipantes(getProfile(), view);
        dto.chat = projetoManager.loadChat(getProfile(), view);
     }
     return dto;
  }
```

Fonte: elaborada pelo autor

Na figura 8 , podemos ver o *controller* que gerencia as ações disponíveis para a tela de projetos. A anotação RestController na assinatura da classe define que ela receberá requisições externas atráves de métodos HTTP (GET, POST, DELETE, PUT, etc). A anotação RequestMapping define a URI principal onde as requisições serão feitas. Cada método público possui seu próprio endereço. Por exemplo, o método load pode ser acessado através da URL "/projeto/{key}", onde "{key}" é a chave do projeto que será acessado. O método no caso é um GET, que apenas entrega os dados a quem fez a requisição. Os métodos dos *controllers* sempre devem ser o mais simples possível, jogando toda a responsabilidade de regras de negócio para os *managers*. No caso dos projetos, toda a lógica está na classe ProjetoManager, que tem como função carregar, persistir e validar os dados vindos do ProjetoController.

Figura 9 - Método save da classe ProjetoManager

```
public ProjetoView save(Profile profile, ProjetoView view) throws ValidationException {
  validate(view);
  Projeto entity;
  if (view.getId() == null) {
     //Cadastrar projeto
     entity = new Projeto();
     entity.setOwner(getAccountLogada(profile));
     entity.setTitulo(view.getTitulo());
     entity.setDescricao(view.getDescricao());
     entity = super.save(entity, profile);

     view.addMessage(new MessageSuccess("success.save"));
  } else {
     //Update
     entity = projetoService.selectById(Projeto.class, view.getId());
     if (!entity.getOwner().getId().equals(getAccountLogada(profile).getId())) {
        throw new ValidationException(new Message("warn.save"));
     }
     entity.setTitulo(view.getTitulo());
     entity.setDescricao(view.getDescricao());
     entity = super.update(entity, profile);

     view.addMessage(new MessageSuccess("success.update"));
  }

  return view;
}
```

Fonte: elaborada pelo autor

Nos *managers*, temos toda a lógica da aplicação. Todas as regras de negócio, como regras de recuperação de dados do banco, criação de objetos para serem persistidos, validação de dados, ficam nestas classes. Além disso, no *manager* é feita a conversão dos DAOs para DTOs, que serão expostos pela API, e vice-versa. Toda a interação com o PostgreSQL é feita nos *services*, que são as classes responsáveis por recuperar, excluir e persistir efetivamente as informações da aplicação.

Vamos usar o método *save* da classe ProjetoManager exibido na figura 9 como exemplo. Este método é invocado pelo *controller*, e tem como função salvar ou atualizar um projeto. Ao entrar no método, a função *validate* é invocada. Esta tem como responsabilidade validar todos os dados do DTO, confirmando que todos são válidos e que a conversão para DAO pode ser feita para que as informações sejam persistidas no banco de dados. Após a validação, uma verificação é executada: se o projeto já possui um *id*, significa que estamos querendo efetuar uma atualização de um projeto já existente em nossa base; se não o possui, estamos criando um novo projeto. Após todas estas etapas, o papel de persistir efetivamente os dados é do *service*, que assim como os *controllers*, possui pouquíssimas regras de negócio.

Figura 10 - ProjetoService

```
@Service
public class ProjetoService extends BaseService {

  public List<Projeto> selectAllByAccountId(Long id) {
     String QUERY = "SELECT obj FROM Projeto obj WHERE obj.owner.id = ?1";
     return super.selectByQuery(QUERY, id);
  }

  public List<Projeto> selectAllQueParticipoByAccountId(Long id) {
     String QUERY = "SELECT obj.projeto FROM AccountProjeto obj WHERE obj.account.id = ?1";
     return super.selectByQuery(QUERY, id);
  }

  public List<Projeto> selectProjetoBySeach(String search) {
     search = search.toLowerCase();
     String QUERY = "SELECT obj FROM Projeto obj WHERE " +
           "LOWER(obj.titulo) LIKE '%" + search + "%' OR " +
           "LOWER(obj.descricao) LIKE '%" + search + "%'";
     return super.selectByQuery(QUERY);
  }
}
```

Fonte: elaborada pelo autor

Os *services* contêm métodos exclusivamente de acesso ao banco de dados. Estes acessos são realizados através de consultas (*queries*), muito parecidas com o padrão utilizado pelos bancos de dados SQL, chamadas de HQL (*Hibernate Query Language*). A HQL é desenvolvida pelo *framework* Hibernate, é totalmente orientada a objetos e se utiliza de muitas vantagens desta abordagem de programação, como herança, polimorfismo e associação.

No método *selectAllByAccountId* da classe ProjetoService exibido na figura 10, temos o exemplo de uma query HQL:

"SELECT obj FROM Projeto obj WHERE obj.owner.id = ?1"

Esta expressão retorna uma lista com todos os projetos criados pelo usuário que possui o *id* passado como parâmetro.

Figura 11 - Entidade de Projeto

```
@Entity
@Table(name="projeto")
public class Projeto extends BaseEntity implements Serializable {

  @Id
  private Long id;

  private StatusProjeto status;

  private String titulo;

  @Column(columnDefinition="TEXT")
  private String descricao;

  @Column(name = "habilidades", columnDefinition="TEXT")
  @Convert(converter = HabilidadesConverter.class)
  private List<HabilidadeBean> habilidadeList;

  @Column(name = "links_externos", columnDefinition="TEXT")
  @Convert(converter = SocialNetworkConverter.class)
  private List<SocialNetworkBean> linksExternos = new ArrayList<>();

  private Boolean permiteRequest = true;

  @Column(name = "tags", columnDefinition="TEXT")
  @Convert(converter = TagsConverter.class)
  private List<TagBean> tags = new ArrayList<>();

  @ManyToOne
  @JoinColumn(name="fk_account")
  private Account owner;
```

```
  @OneToMany(mappedBy="projeto")
  private List<AccountProjeto> accountProjetoList;

  @OneToMany(mappedBy = "projeto", fetch = FetchType.LAZY)
  private List<Post> postList;

  @Column(name = "mensagens", columnDefinition="TEXT")
  @Convert(converter = MensagemConverter.class)
  private List<MensagemBean> mensagens = new ArrayList<>();

  private String chave;
  @Column(name = "insert_time")
  private Timestamp insertTime;
  @Column(name = "update_time")
  private Timestamp updateTime;
  @Column(name = "insert_by")
  private Long insertBy;
  @Column(name = "update_by")
  private Long updateBy;

  public Projeto() {

  }

  public enum StatusProjeto {
     ATIVO, INATIVO, REMOVIDO
  }

  public MensagemBean getMensagemById(Long id) {
     return this.mensagens.stream()
           .filter(m -> m.getId().equals(id))
           .findAny()
           .orElseGet(null);
  }

  public boolean deleteMensagemById(Long id) {
     return this.mensagens.removeIf(l -> l.getId().equals(id));
  }

  public AccountProjeto getAccountProjetoByAccountId(Long id) {
     if (this.accountProjetoList == null) {
        return null;
     }

     return this.accountProjetoList.stream().filter(ap -> ap.getAccount().getId().equals(id))
           .findAny().orElse(null);
  }
}
```

Fonte: elaborada pelo autor

A figura 11 mostra resumidamente a entidade que define as características de um projeto na aplicação. As entidades utilizadas para abstrair os dados persistidos na base são anotadas com @Entity, e toda a tabela do banco de dados relacional tem uma entidade que a representa. Todos os relacionamentos encontrados nas tabelas estão representados nas entidades através de anotações. Quando requisitamos uma informação que possui outras relações, estas são automaticamente recuperadas pelo Hibernate, e estão prontas para serem utilizadas pelo programa.

Para facilitar ações que são executadas com frequência na aplicação, algumas classes foram criadas e todos os *services*, *managers* e *controllers* estendem suas respectivas classes base. Precisamos recuperar constantemente informações do usuário que está requisitando e persistindo dados no sistema, para isso foi criado um método chamado *getAccountLogada* na classe BaseManager, figura 12, e todos os managers estendem esta classe e possuem livre acesso a este método. Este é um recurso muito utilizado nas linguagens de programação orientada a objetos e facilita bastante o desenvolvimento, reaproveitamento e manutenção do código escrito.

Figura 12 - TCCBaseManager

```
public abstract class TCCBaseManager extends BaseManager {

  @Inject
  private ProfileSingleton profileSingleton;

  public ProfileSingleton getProfileSingleton() {
     return profileSingleton;
  }

  private Account getUsuario(Profile profile) {
     return (Account) profile.getUsuario();
  }

  public Account getAccountLogada(Profile profile) {
     return super.load(Account.class, getUsuario(profile).getId());
  }

  public Account getAccountLogadaLoaded(Profile profile) {
     return super.load(Account.class, getUsuario(profile).getId());
  }
  public File saveArquivoNoDisco(File file, InputStream inputStream, boolean image) {
     OutputStream outputStream = null;
     try { //FileNotFoundException
        file.createNewFile();
        outputStream = new FileOutputStream(file);
```

```
            IOUtils.copy(inputStream, outputStream);
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            IOUtils.closeQuietly(outputStream);
            IOUtils.closeQuietly(inputStream);
        }
        return file;
    }
}
```

Fonte: elaborada pelo autor

A totalidade do projeto desenvolvido segue os padrões organizacionais e de nomenclatura mostrados nas figuras acima. Deste modo, qualquer desenvolvedor com um pouco de experiência em aplicações web pode, em poucas horas, ter domínio completo sobre o sistema e adicionar novas funcionalidades.

5.1.1 Endpoints

Um *endpoint* é simplificadamente um recurso da aplicação exposto para que possa ser consumido por clientes externos. No caso de *web services RESTful*, a exposição é feita através de *URIs.* Uma *URI (Universal Resource Identifier)*, é "uma sequência de caracteres que identifica um recurso lógico ou físico".

Na figura 13, exibida abaixo, estão listadas todas as *URIs* disponíveis no servidor para serem consumidas pelo cliente da aplicação. Todos os métodos do tipo *POST* consomem um objeto do tipo *JSON* e todos os métodos produzem uma resposta em um objeto também do tipo *JSON*.

Figura 13 - Endpoints

LoginController - Métodos que tratam do login e criação de conta
POST - /login Faz login com usuário e senha
POST - /facebook-login Faz login pelo Facebook
POST - /linkedin-login Faz login pelo LinkedIn

POST - /logoff Faz logoff
POST - /signin Faz o cadastro de um novo usuário
AccountController - Edição de conta do usuário
GET - /account Recupera informações da conta do usuário
GET - /account/avatar/{key} Recupera a foto de um usuário através da chave
POST - /account/save-avatar Atualiza a foto do usuário
POST - /account/update Atualiza as informações (nome, email, senha) do usuário
ChatController - Gerencia as mensagens do usuário
GET - /chat Recupera a lista de chats do usuário
GET - /chat/{id} Recupera um chat do usuário através do id
POST - /chat Salva uma mensagem
POST - /chat/delete-message Exclui um mensagem em um chat
ContactsController - Gerencia os contatos do usuário
GET - /contact Recupera a lista de contatos do usuário
GET - /contact/request Pede para adicionar um usuário em sua lista de contatos
GET - /contact/accept/{key} Aceita um pedido de amizade através do id do usuário que solicitou
GET - /contact/ignore/{key} Ignora um pedido de amizade através do id do usuário que solicitou
GET - /contact/cancel/{key} Cancela um pedido de amizade
DELETE - /contact/{key} Exclui um usuário da lista de contatos
LocationController - Busca cidades no banco de dados
GET - /location/cidade Pesquisa as cidades com base no nome informado
NotificationController - Gerencia as notificações do usuário
GET - /notifications Recupera as notificações do usuário
POST - /notifications Marca como lidas todas as notificações passadas no método
ProfileController - Recupera perfis de usuários do banco de dados
GET - /profile/{key} Faz o load de um perfil através da chave do usuário
ProfileEditController - Edição do perfil do usuário
GET - /profile-edit/initial-data Carrega todas as informações necessárias para editar um usuário
GET - /profile-edit Carrega o perfil do usuário da sessão

POST - /profile-edit Salva o perfil do usuário da sessão
PostController - Gerencia os posts de um projeto
POST - /post Salva um novo post ou atualiza um post existente em um projeto
DELETE - /post Remove o post de um projeto
POST - /post/save-comment Salva um comentário em um post
POST - /post/delete-comment Exclui um comentário em um post
ProjectController - Gerencia os projetos
GET - /project List os projetos do usuário da sessão
GET - /project/{key} Faz o load de um projeto através de sua chave
POST - /project Salva um novo projeto ou atualiza um existente
POST - /project/request Usuário da sessão pede para entrar em um projeto
POST - /project/accept Usuário é aceito em um projeto
POST - /project/delete-participante Exclui um usuário de um projeto
POST - /project/save-mensagem-to-owner Envia um mensagem para o dono de um projeto
POST - /project/save-mensagem Envia uma mensagem para todos os participantes do projeto
POST - /project/delete-mensagem Exclui uma mensagem do projeto
SearchController - Busca projetos
POST - /search Busca um projeto no banco de dados

Fonte: elaborada pelo autor

5.2 Front-end

Conforme já descrevemos na seção front-end do capítulo de fundamentos tecnológicos, a interface de nossa aplicação será desenvolvida utilizando HTML, CSS e o framework AngularJS, desenvolvido pela Google e escrito em JavaScript.

> "Criado por Mišo Hevery e Adam Abrons em 2009, o AngularJS é um framework JavaScript que promove uma experiência de desenvolvimento de aplicações web com alta produtividade.
> Foi construído baseado na crença de que a programação declarativa é a melhor escolha para o desenvolvimento da interface do usuário, enquanto a programação imperativa é muito melhor para a implementação da lógica de negócio.
> Para alcançar isso, o AngularJS estende o HTML tradicional e seu vocabulário atual, facilitando a vida dos programadores.
> O resultado é o desenvolvimento de código altamente reutilizável e de fácil manutenção através de seus componentes, deixando para trás horas de código desnecessário e mantendo o time focado em coisas mais importantes." (BRANAS, 2014, página 25)

O AngularJS utiliza o recurso de *Single-page Application*, que utiliza uma única página web para exibir a aplicação inteira, promovendo a experiência do usuário e fazendo com que o sistema se pareça com uma aplicação desktop.

Nas SPAs, todo o código principal do programa, como HTML, CSS e JavaScript, é carregado junto da primeira requisição, e outros módulos são carregados dinamicamente conforme necessidade, normalmente em resposta à ações requisitadas pelo usuário. A página nunca é recarregada, apenas as *views* são renderizadas de acordo com a tela que o cliente está querendo acessar.

Assim como no back-end, o front-end de nossa aplicação também está organizado de forma com que os desenvolvedores possam localizar facilmente os arquivos e recursos do programa, tornando simples a manutenção do mesmo. A aplicação na parte do cliente está dividida basicamente em *views*, controladores, serviços e diretivas. A *view* nada mais é do que a página que o usuário está vendo; um trecho de código HTML que define o comportamento e template de uma tela do programa. O acesso às *views* é definido no arquivo *routes.js* e nele estão todas as configurações de cada *view*. Na figura a seguir, temos o *routes.js* de nosso projeto:

Figura 13: O routes.js

```
app.config(function($routeProvider) {
  $routeProvider.when('/home', {templateUrl: '/home.html', controller: 'homeCtrl as vm'});
  $routeProvider.when('/contatos', {templateUrl: '/contatos.html', controller: 'contatosCtrl as vm'});
  $routeProvider.when('/mensagens', {templateUrl: '/mensagens.html', controller: 'mensagensCtrl as vm'});
   $routeProvider.when('/my-account/', {templateUrl: '/my-account.html', controller: 'myAccountCtrl as vm'});
   $routeProvider.when('/profile-edit/', {templateUrl: '/profile-edit.html', controller: 'profileEditCtrl as vm'});
   $routeProvider.when('/profile/:key', {templateUrl: '/profile.html', controller: 'profileCtrl as vm'});
   $routeProvider.when('/search/:key', {templateUrl: '/search.html', controller: 'searchCtrl as vm'});
   $routeProvider.when('/my-projects', {templateUrl: '/my-projects.html', controller: 'meusProjetosCtrl as vm'});
   $routeProvider.when('/new-project', {templateUrl: '/new-project.html', controller: 'novoProjetoCtrl as vm'});
   $routeProvider.when('/edit-project/:key', {templateUrl: '/new-project.html', controller: 'novoProjetoCtrl as vm'});
   $routeProvider.when('/projeto/:key', {templateUrl: '/projeto.html', controller: 'projetoCtrl as vm'});

  //other
  $routeProvider.otherwise({redirectTo: '/home'});
});
```

Fonte: elaborada pelo autor

Quando o usuário acessa o contexto "/contatos", a template utilizada para carregar as informações, ou seja, o trecho de código HTML exibido na tela, está no arquivo "contatos.html"; e o *controller* responsável por esta *view* é o contatosCtrl. Toda a implementação do front-end segue uma nomenclatura padrão: os nomes dos arquivos indicam explicitamente o objetivo e função dos mesmos. Isso também ajuda na manutenibilidade e na localização eficiente de partes do código.

Ao carregar a tela de contatos o arquivo “contatos.html” é renderizado e exibido no navegador do usuário. No controlador contatosCtrl, responsável por esta *view*, estão todas as funções que definem o comportamento desta tela.

Os *controllers* de uma aplicação AngularJS são responsáveis por fazer a ponte entre a interface do usuário e a lógica de negócios do *front-end.* Um controlador Angular define todos os comportamentos da tela: o que acontece quando o usuário clica em um

determinado botão, quando submete um formulário, o que a página deve carregar quando é acessada, etc.

Na figura 14 é mostrado o código referente ao *mensagensCtrl* do sistema, responsável por gerenciar o comportamento da tela de mensagens da aplicação.

Figura 14 - Exemplo de um *controller*

```
app.controller("mensagensCtrl", ['usuarioLogadoService', "mensagensService", function
(usuarioLogadoService, mensagensService) {
  var vm = this;

  vm.usuarioLogado = {};
  vm.chats = [];
  vm.chatLoaded = {};

  //functions
  vm.loadChat = loadChat;
  vm.saveMensagem = saveMensagem;
  vm.removeMensagem = removeMensagem;

  function loadChat(chat) {
     mensagensService.load(chat.id).success(function(view) {
        console.log(view);
        vm.chatLoaded = view;
     });
  };

  function saveMensagem(mensagem) {
     mensagem.chatId = vm.chatLoaded.id;
     mensagensService.saveMensagem(mensagem).success(function(view) {
           mensagem = undefined;
           vm.chatLoaded.mensagens.push(view);
       });
  };

   function removeMensagem(mensagem) {
     var index = vm.chatLoaded.mensagens.indexOf(mensagem);
       mensagem.chatId = vm.chatLoaded.id;
       mensagensService.removeMensagem(mensagem).success(function() {
           vm.chatLoaded.mensagens.splice(index, 1);
       });
   };

  var iniciarTela = function() {
     vm.usuarioLogado = usuarioLogadoService.getUsuario();
     mensagensService.list().success(function(data) {
        console.log(data);
        vm.chats = data;
     });
  };
```

```
   iniciarTela();
}]);
```

Fonte: elaborada pelo autor

Ao acessar a tela, o método "iniciarTela" é executado, fazendo uma chamada para o *mensagensService*, que por sua vez faz uma requisição do tipo GET para o servidor REST, pedindo todas as conversas do usuário logado no sistema. Esta ação serve para carregar os dados iniciais em tela, ou seja, a lista de mensagens recebidas pelo usuário, de forma com que ele possa carregar um *chat* e responder mensagens.

Todos as funções que iniciam com a expressão *vm* servem para interação com algum elemento da interface. Por exemplo, quando o usuário seleciona uma conversa na lista carregada anteriormente pela função "iniciarTela", o método "loadChat" é invocado, recebendo o *id* do chat que o usuário deseja ver, e passando a requisição para o serviço responsável, que neste caso é o *mensagensService*. A resposta retornada do servidor são as mensagens enviadas e recebidas na conversa selecionada, e os dados são carregados na tela.

No exemplo, percebemos que apesar do controlador ser responsável por toda interação e comportamento da interface do sistema, quem envia e recebe as requisições HTTP do servidor *back-end* é o *service*. Na figura 15 podemos ver o código do serviço que gerencia as chamadas da tela de mensagens: o *mensagensService*.

Figura 15 - *mensagensService*

```
app.factory('mensagensService', ['ajaxService', function(ajaxService) {
  var _pageUrl = "/chat";

  var _initialData = function() {
     return ajaxService.get(_pageUrl+"/initial-data");
  };
  var _list = function() {
     return ajaxService.get(_pageUrl);
  };
  var _load = function(id) {
      return ajaxService.get(_pageUrl + "/" + id);
  };
  var _saveMensagem = function(view) {
      return ajaxService.post(_pageUrl, view);
  };
  var _removeMensagem = function(view) {
      return ajaxService.post(_pageUrl + "/delete-mensagem", view);
```

```
  };

  return {
     initialData : _initialData,
     list : _list,
     load : _load,
     saveMensagem : _saveMensagem,
     removeMensagem : _removeMensagem
  }

}]);
```

Fonte: elaborada pelo autor

Na grande maioria dos casos, um serviço não possui qualquer tipo de lógica de programação, ou seja, seu objetivo é puramente enviar requisições HTTP e receber a resposta do servidor. O método *load* é utilizado no controlador mostrado anteriormente para carregar as mensagens de uma determinada conversa na tela. Ele faz uma requisição do tipo GET para a URL /chat do servidor *back-end*, que trata de recuperar as mensagens desta conversas no banco de dados, tratá-las e enviá-las ao cliente.

Um recurso bastante interessante utilizado no AngularJS é a criação de diretivas. Com diretivas você pode estender código HTML com elementos e atributos. É possível utilizar esta técnica em diversas situações. Por exemplo, você pode definir sua própria tag para descrever uma parte da aplicação, especializando e encapsulando o código HTML e a lógica de forma com que seja simples reutilizá-los em várias partes do sistema.[12]

A diretiva mais expressiva e talvez a mais importante criada na aplicação desenvolvida neste trabalho, foi a de notificações. Através da tag ***<notifications>***, podemos mostrar uma listagens de todas as notificações recebidas pelo usuário logado em qualquer tela do sistema. Nas figuras 16 e 17, temos a definição desta diretiva e de sua template HTML.

Figura 16 - Definição da diretiva de notificações

```
app.directive('notifications', ['$interval', 'notificationsService', function($interval,
notificationsService) {
 return {
   restrict: 'E',
   scope: { },
   replace : true,
   templateUrl: '/templates/notifications.html',
   link: function (scope) {
     scope.notifications = [];
     scope.notificationsCount = 0;

     //functions
     scope.readAll = readAll;

     function readAll() {
       var ids = [];
       angular.forEach(scope.notifications, function(obj) {
         obj.read = true;
         ids.push(obj.id);
       });

       var view = {ids: ids};
       notificationsService.read(view);
     };




     //Notificacoes
     var getNotifications = function() {
         notificationsService.list().success(function(notifications) {
             scope.notifications = notifications;
             scope.notificationsCount = 0;
             angular.forEach(scope.notifications, function(obj) {
                 if (!obj.read) {
                     scope.notificationsCount += 1;
                 }
             });
         });
     };

     getNotifications(); //Pega no load da pagina
     $interval(getNotifications, 5000); //Refresh a cada 5 segundos
   }
 };
}]);
```

Fonte: elaborada pelo autor

Figura 17 - A template da diretiva de notificações

```
<li class="dropdown notifications updates hidden-xs hidden-sm">
   <a href="" class="dropdown-toggle" data-toggle="dropdown">
       <i class="fa fa-bell-o"></i>
       <span class="badge badge-primary">{{notificationsCount}}</span>
   </a>
   <ul class="dropdown-menu" role="notification">
       <li class="dropdown-header">Notifications <span class="pull-right"
ng-click="readAll()"><a href="">Mark all read</a></span></li>
       <li class="media" ng-repeat="n in notifications" ng-style="!n.read &&
{'background-color':'#ECF0F1'}">
           <div class="media-left">
               <span class="icon-block s30 bg-grey-200"><i class="fa fa-plus"></i></span>
           </div>
           <div class="media-body">
               <div><a ng-href="{{n.url}}">{{n.description}}</a></div>
               <p>
                   <span class="text-caption text-muted">{{n.insertTime}}</span>
               </p>
           </div>
       </li>
   </ul>
</li>
```

Fonte: elaborada pelo autor

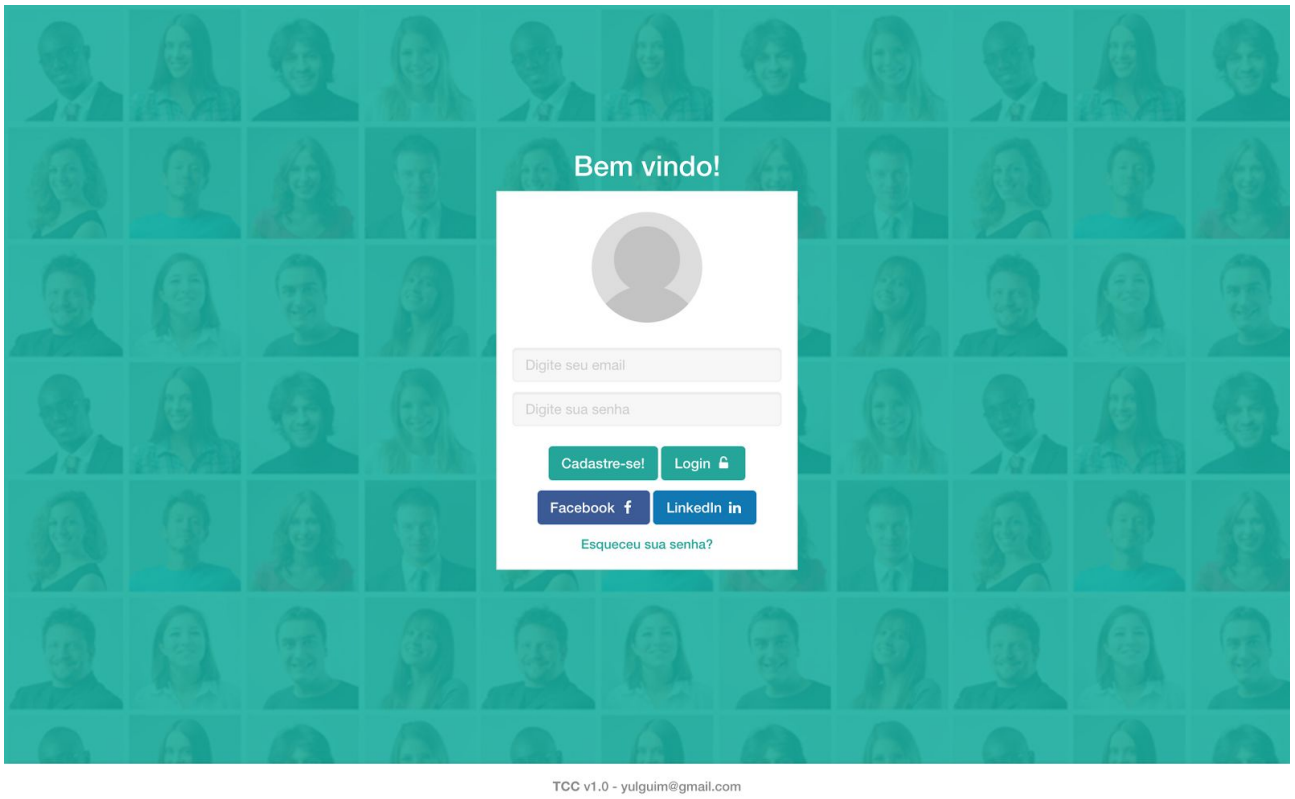
Esta é uma diretiva simples e direta que exemplifica bem sua função: mostrar as notificações do usuário. Quando ela é carregada, o método *getNotifications*() é invocado, e logo após ele é criado um *$interval*, que faz com essa função seja chamada infinitamente de 5 em 5 segundos, para que caso uma nova notificação tenha sido criada, ela seja mostrada na tela sem que o usuário precise recarregar a página. O método chama o serviço responsável por recuperar as notificações do servidor e monta uma lista com todas elas e faz a contagem das notificações ainda não lidas pelo usuário. Estas informações são carregadas na lista presente na *template* HTML desta diretivas, e então exibidas na tela. Na *template* também existe um botão para marcar as mensagens ali mostradas como lidas. Quando o usuário clica neste botão, o método *readAll*() é chamado, iterando todas as notificações e marcando o atributo *read* como *true*. Estas notificações são enviadas ao servidor, que persiste as informações no banco de dados.

É interessante ressaltar que, assim como no *back-end* da aplicação, o *front-end* também segue um padrão de nomenclatura de arquivos e distribuição de responsabilidades bem estabelecido, e todas as telas seguem o padrão exibido nos

exemplos acima. Desta forma, qualquer pessoa com conhecimento em desenvolvimento *JavaScript* pode estender e criar novos recursos no módulo do cliente.

5.2.1 A interface da aplicação

Figura 18 - Login

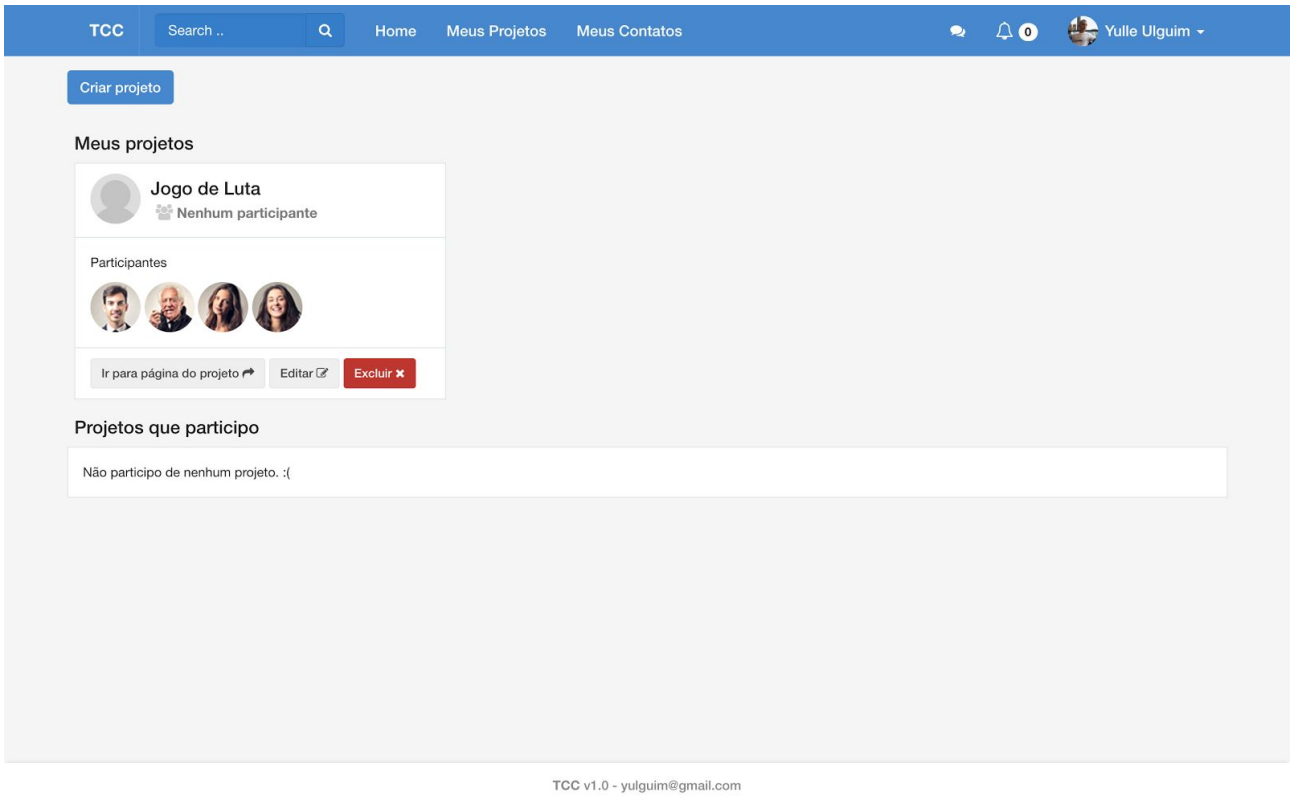


Fonte: elaborada pelo autor

Quando o usuário acessa o sistema, a tela de login, na figura 18, é a primeira tela que ele vê. Nesta tela, é possível efetuar login na aplicação utilizando email e senha, ou fazer login através do Facebook[13] ou LinkedIn[14]. Para pessoas que ainda não possuem conta, é possível criar uma conta utilizando um email válido.

[13] http://www.facebook.com
[14] http://www.linkedin.com

Figura 19 - Meus projetos

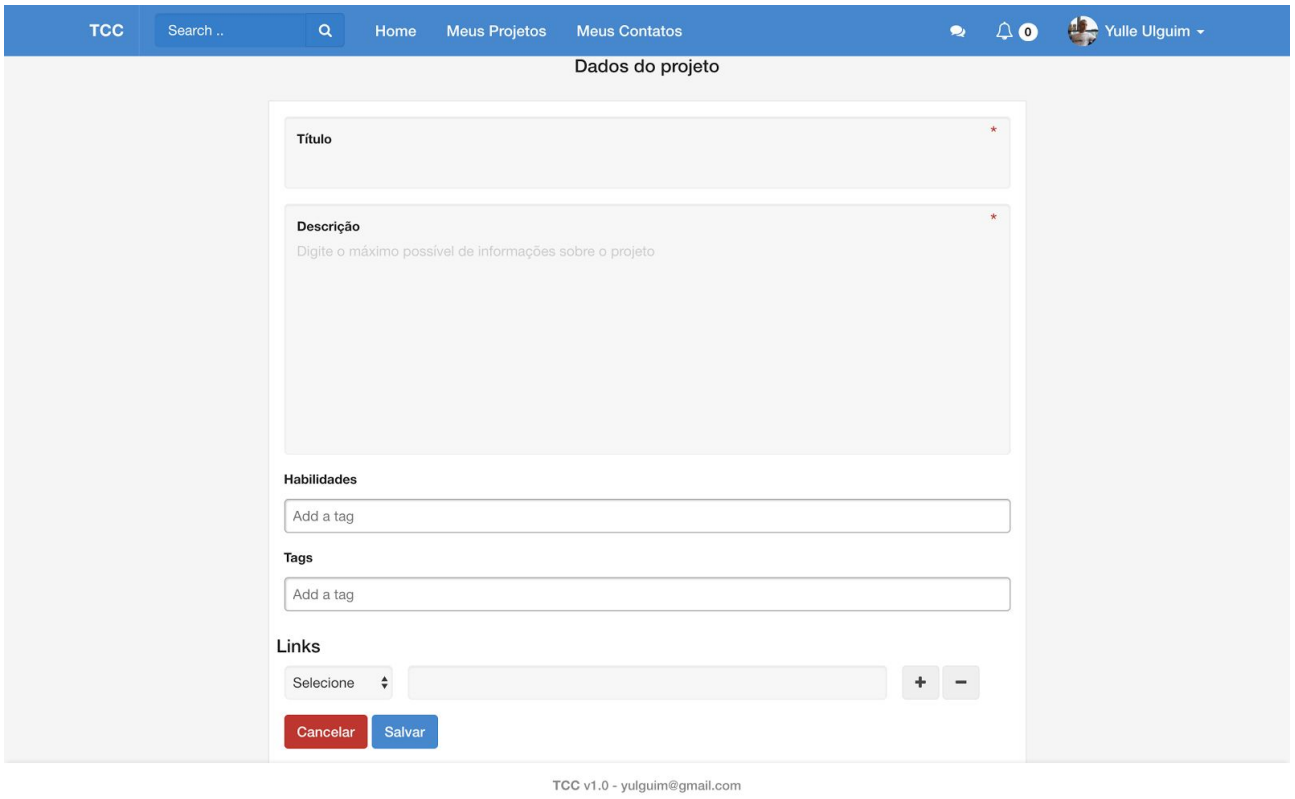


Fonte: elaborada pelo autor

Na figura 19, a tela "meus projetos" exibe todos os projetos criados pelos usuário, além de todos os projetos dos quais o usuário participa. Nesta página também é possível excluir um projeto e acessar as páginas de criação de novos projetos e de edição de projetos através de *links*.

A figura 20 mostra a tela de criação e edição de projetos. Quando um usuário clica no botão de criar novo projeto, a página é carregada em branco e os dados do projeto a ser criado são inseridos. Se a ação escolhida é editar um projeto já existente, os dados deste são carregados nesta mesma tela, e o usuário pode editar as informações recuperadas do banco de dados.

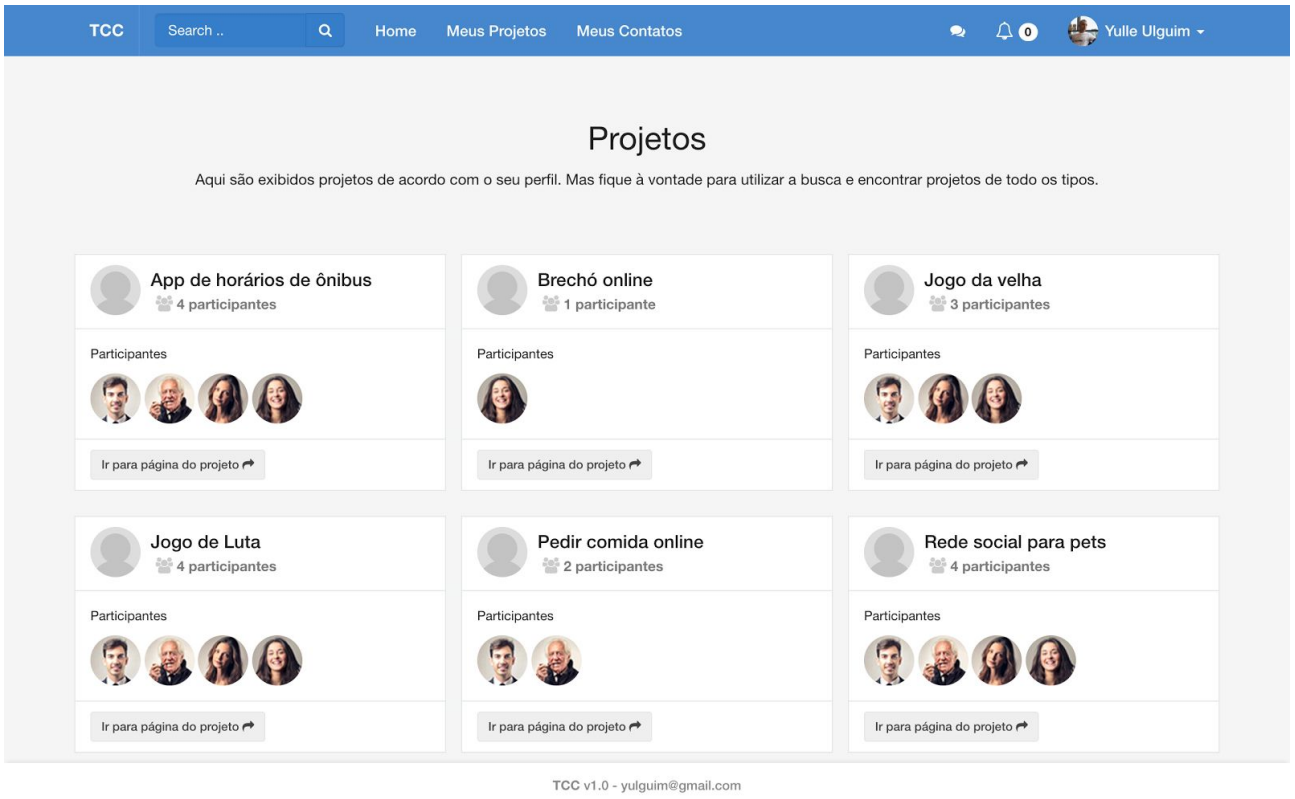
Figura 20 - Criação/edição de projetos



Fonte: elaborada pelo autor

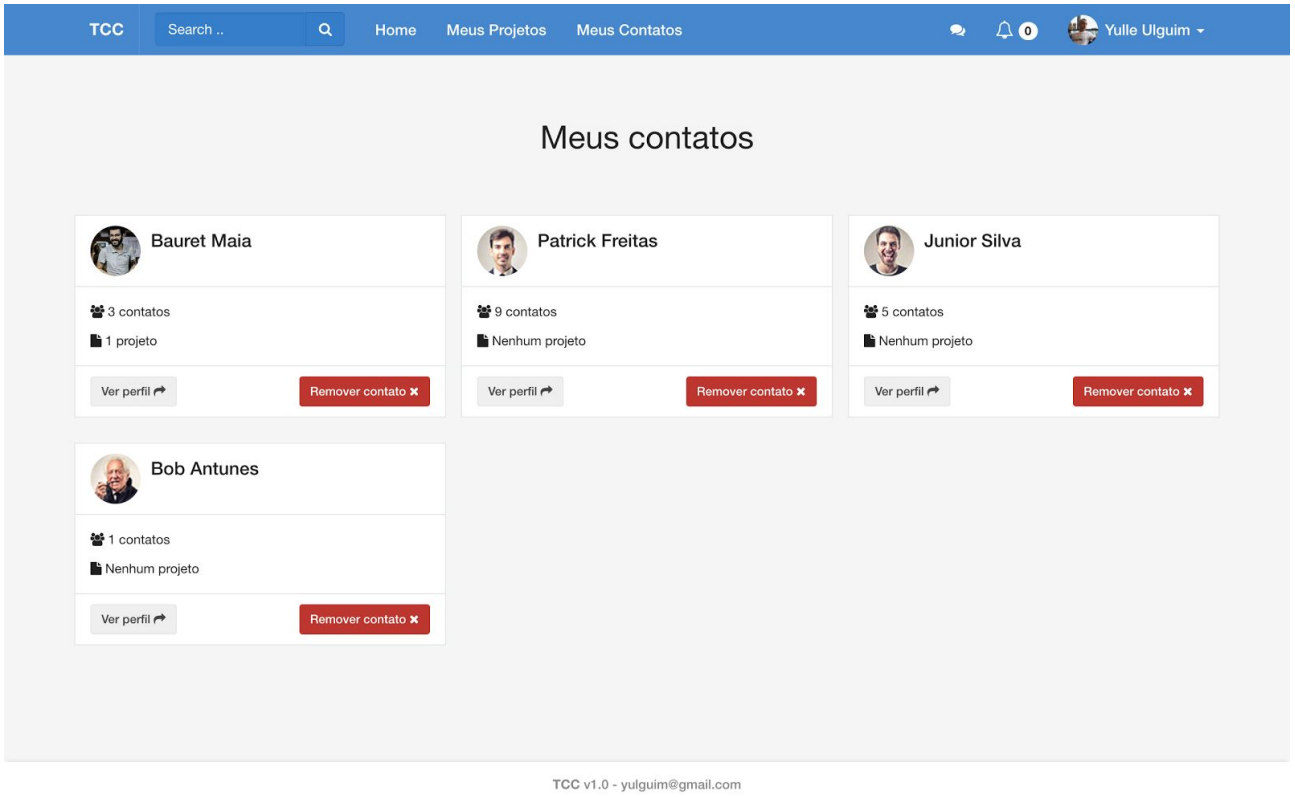
Na tela principal da aplicação, chamada de *home*, são mostrados os projetos disponíveis que se encaixam no perfil do usuário logado. Se o usuário é por exemplo, programador C# e entende de *Adobe Photoshop*, serão exibidos apenas projetos que se encaixam nestas habilidades. Esta página facilita o descobrimento de novos projetos sem que seja preciso utilizar a pesquisa, aumentando a possibilidade de que novos usuários participem e interajam com a aplicação e com projetos de forma rápida e simples. Na figura 21, temos o exemplo da tela *home* de um usuário do sistema.

Figura 21 - Página *home*



Fonte: elaborada pelo autor

Figura 22 - Meus contatos

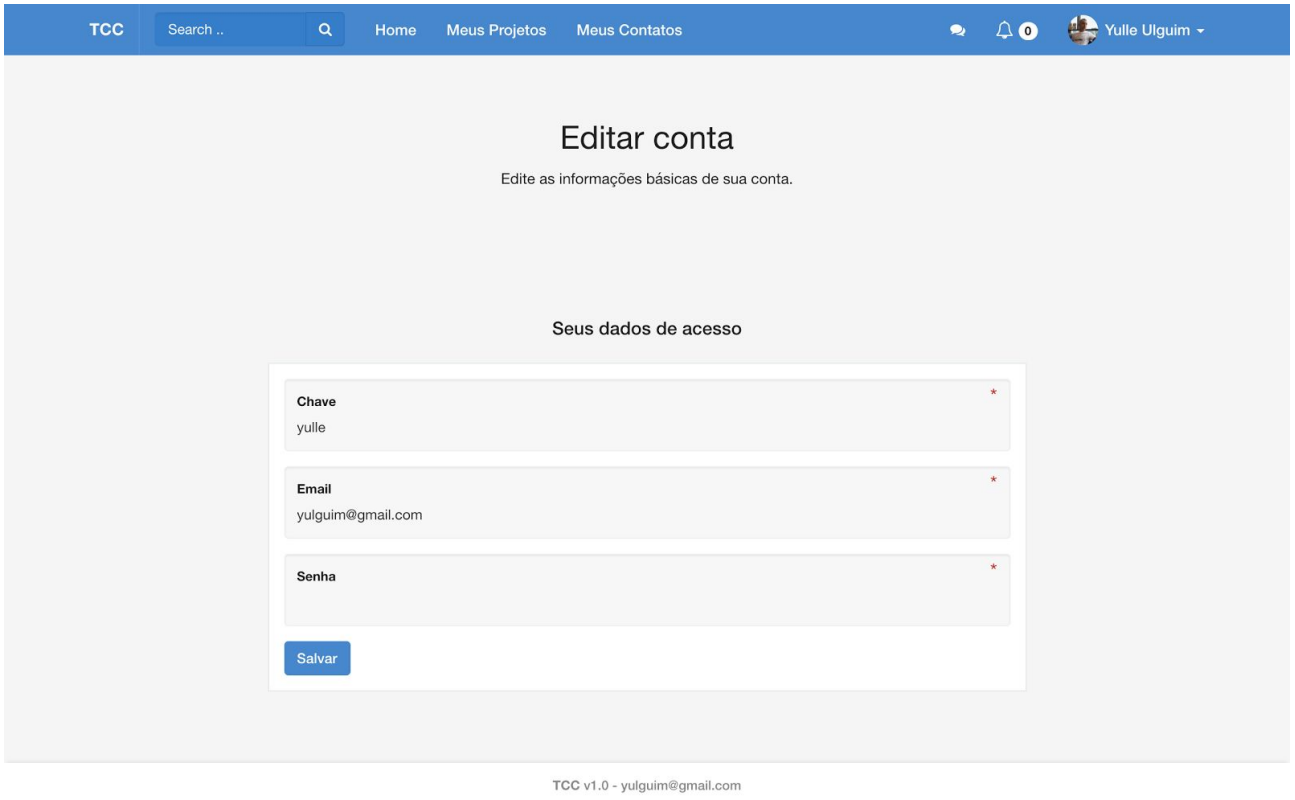


Fonte: elaborada pelo autor

A página "meus contatos", mostrada na figura 22, exibe todos os contatos do usuário e algumas informações adicionais, como número de contatos de cada usuário e número de projetos em que o usuário está. Além disso, nesta tela é possível acessar o perfil de cada contato e também remover o contato da sua rede de relacionamentos.

A edição de informações básicas de uma conta, como e-mail, senha e chave do usuário, que é mostrada no *link* de acesso para seu perfil, é feita através da página "editar conta". A figura 23 mostra um exemplo deste tela.

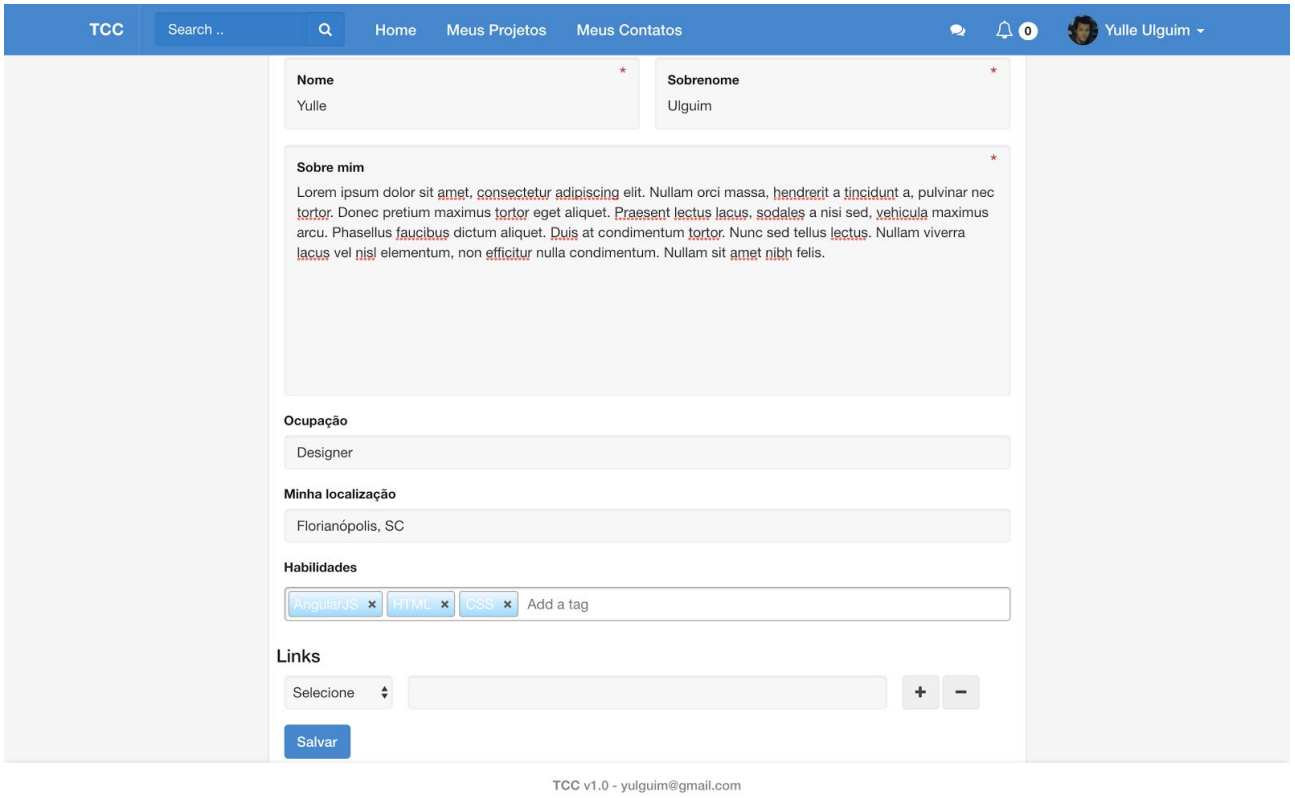
Figura 23 - Editar conta



Fonte: elaborada pelo autor

Na tela "editar perfil", mostrada na figura 24, é possível atualizar todas as informações que são mostradas no perfil público do usuário. Nome, sobre mim, ocupação, habilidades e cidade onde reside são definidos nesta página. Além disso, é possível alterar sua foto de perfil e adicionar links para redes sociais.

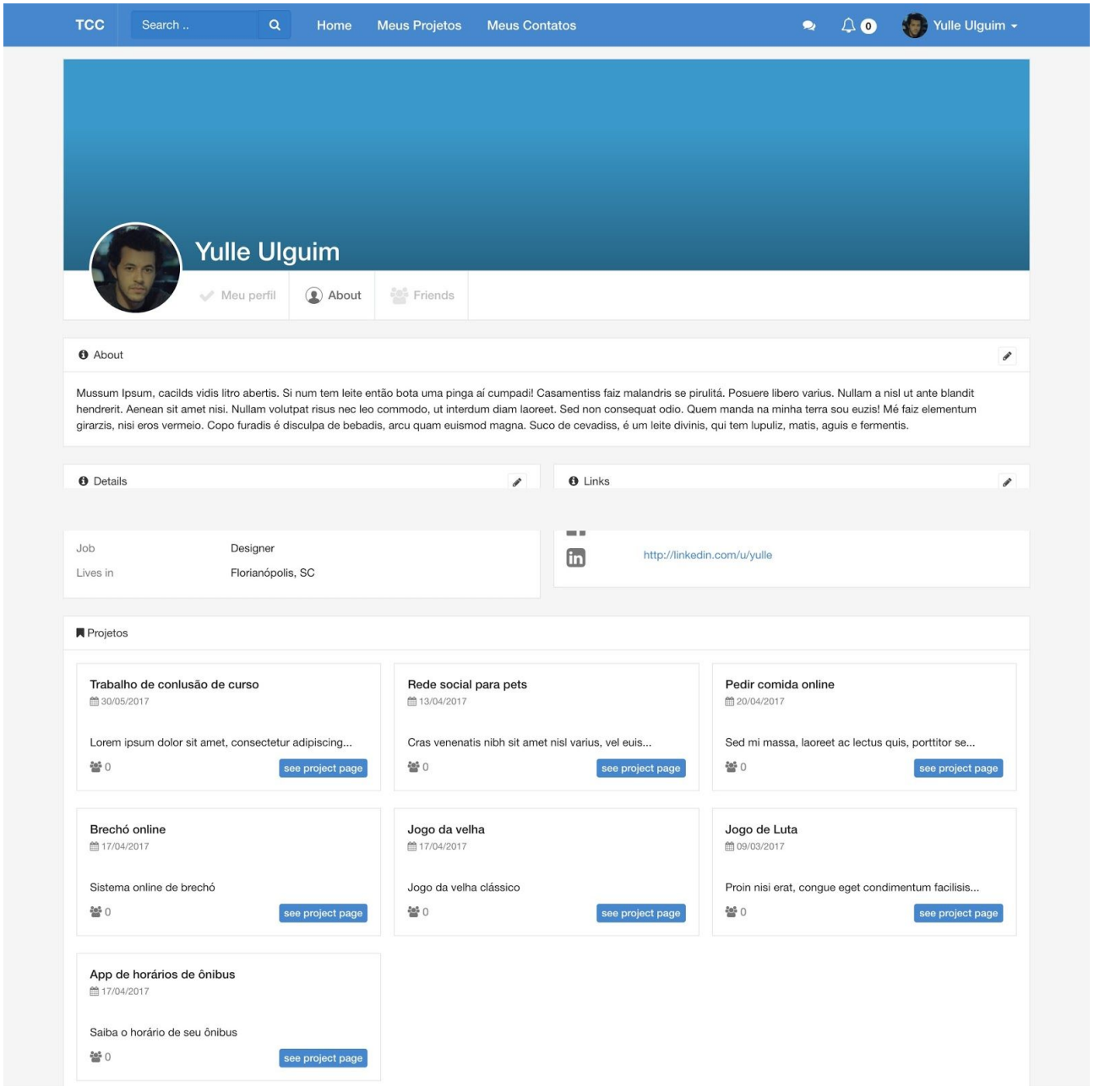
Figura 24 - Editar perfil



Fonte: elaborada pelo autor

A tela de perfil exibe as informações definidas na tela "editar perfil" para que todos os utilizadores da plataforma tenham acesso aos dados de outros usuários. Através desta página é possível ver as habilidades, links externos e projetos dos quais o usuário participa. Esta pode ser considerada, ao lado da página de projeto, a tela mais importante do sistema, afinal é o cartão de visita de uma pessoa que está procurando projetos para participar. A figura 25 mostra um perfil público na aplicação.

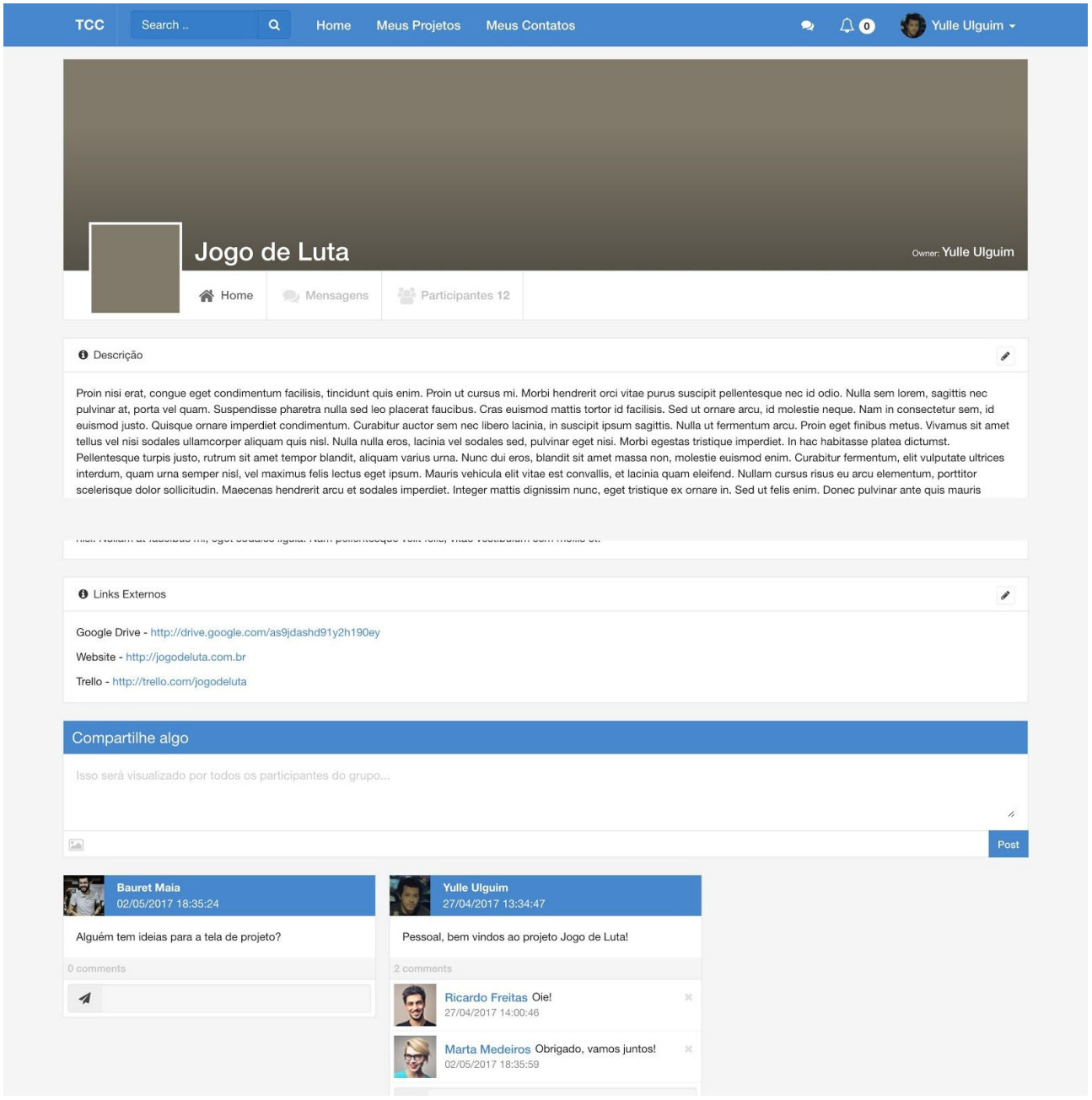
Figura 25 - Tela de perfil público



Fonte: elaborada pelo autor

Por último, temos a tela de projeto. Todas as informações de um projeto são exibidas nesta tela. Além disso, toda a interação entre os usuário que participam do projeto é feita diretamente nesta tela. Alguns recursos, como envio de mensagens, só são liberados para para usuários aceitos pelo dono do projeto. O proprietário também pode criar *posts*, onde todos os participantes podem comentar e interagir. São exibidos também *links* para documentos ou aplicações externas utilizadas para ajudar no desenvolvimento do projeto. Na figura 26, vemos um exemplo de um projeto chamado "Jogo de Luta".

Figura 26 - Tela de projeto



Fonte: elaborada pelo autor

5.3 Implantação

Para hospedar a aplicação na *web* foi criada uma instância na nuvem de uma máquina virtual utilizando a infra-estrutura da *DigitalOcean*[15]. O servidor possui configurações simples mas suficientes para rodar o sistema: processador com 1 núcleo, 1Gb de memória RAM e 30 Gb de SSD. O sistema operacional escolhido foi o CentOS 7[16] , rodando a versão 9.6 do PostgreSQL[17], Java 8[18], e o servidor *web* Nginx[19] versão 1.12.

A aplicação se encontra disponível para acesso em http://tcc.carinae.in/

[15] http://www.digitalocean.com
[16] http://www.centos.org
[17] http://www.postgresql.org
[18] http://www.java.com
[19] http://www.nginx.org

6. Conclusão

A motivação para o desenvolvimento desta ferramenta se deu pelo fato de não existir uma opção como esta disponível na internet. Todas as aplicações existentes atualmente têm como objetivo um relacionamento estritamente profissional e de curto prazo entre o dono de um projeto e um profissional *freelancer*, que quase sempre assume um determinado trabalho tendo em vista a sua remuneração.

Espero que este trabalho possa preencher a carência de um nicho pouco explorado, visando fomentar a cultura do compartilhamento de ideias, concentrando pessoas interessadas na inovação e no desenvolvimento de novos aplicativos; e porque não, na criação de novas *startups* e empresas.

Durante a etapa do projeto, foram levantados os principais requisitos desejados por este tipo de aplicação para preencher a falta de opções neste nicho de mercado e cumprim com o objetivo final do projeto: reunir usuários com a disposição de colocar ideias e projetos em prática. Na implementação, algumas das tecnologias necessárias para a execução do mesmo já eram conhecidas pelo autor, mesmo que não profundamente. Outras foram estudadas na etapa de implementação, o que tornou a implementação um desafio e gerou algumas dificuldades. Os requisitos fundamentais para que a aplicação funcione da forma que foi idealizada foram cumpridos com sucesso, e a implementação de novas funcionalidades depende apenas de tempo para desenvolvimento e de demanda dos usuários.

Não foram realizados testes intensos de usabilidade e dos recursos disponíveis do sistema nem testes automatizados, mas um pequeno grupo de pessoas próximas do autor utilizou a aplicação por um curto período de tempo, concluindo que a mesma encontra-se em um estágio de desenvolvimento no qual já pode ser colocada em uso.

Por parte do autor, a execução do projeto como um todo foi de grande importância, afinal, no mercado de trabalho é bastante raro ter a oportunidade e a possibilidade de estar presente em todas as etapas do desenvolvimento de um produto, além de poder tomar todas as decisões necessárias para o cumprimento dos objetivos definidos no início do trabalho.

6.1 Trabalhos futuros

É interessante destacar que da maneira que a aplicação foi projetada e desenvolvida, se torna relativamente simples a implementação de novas funcionalidades até mesmo por outros desenvolvedores que não o autor.

Como sugestão de trabalhos futuros, podemos destacar principalmente a execução de testes mais completos com pessoas das mais variadas áreas de atuação e utilizar os resultados para aplicar melhorias no sistema. Também seria interessante a criação de uma identidade visual, de forma com que o produto tenha um design único, uma logo, um nome e seja facilmente reconhecido pelos usuários.

A adição de um recurso como o *crowdfunding* de projetos, onde um usuário da plataforma pode financiar uma ideia mesmo sem participar ativamente dela também é interessante. A integração com ferramentas de gerência de projetos como *Trello*, *Jira*, *Slack* e outras pode ser simples e ter um resultado positivo na adesão de novos colaboradores.

Por último, a implementação de notificações por email ou por aplicativo para *smartphones* se torna indispensável na medida que o número de projetos e de usuários for crescendo, de forma com que um participante possa ser informado de alterações, solicitações, mensagens e novas publicações em um determinado projeto sem precisar acessar a plataforma para checar.

7. Referências

[1] MERRIAM WEBSTER DICTIONARY. Crowdsourcing definition. Disponível em <http://www.merriam-webster.com/dictionary/crowdsourcing>. Acesso em: 04 de junho de 2016.

[2] ANGULARJS. AngularJS. Disponível em <http://angularjs.org>. Acesso em: 06 de junho de 2016.

[3] POSTGRESQL GLOBAL DEVELOPMENT GROUP. PostgreSQL. Disponível em: <http://postgresql.org>. Acesso em: 10 de junho de 2016.

[4] PIVOTAL SOFTWARE. Spring Boot Documentation. Disponível em: <http://projects.spring.io/spring-boot>. Acesso em: 10 de junho de 2016.

[5] VALA, Andre. Working with AngularJS. Disponível em: <http://www.slideshare.net/AndreVala/working-with-angularjs>. Acesso em 15 de junho de 2016.

[6] VOSS, Jakob; HORN, Moritz. Exposing Library Services with AngularJS. 2014. Disponível em: <http://journal.code4lib.org/articles/10023>. Acesso em 20 de junho de 2016.

[7] WIKIPEDIA. Size Comparisons. Disponível em: <https://en.wikipedia.org/wiki/Wikipedia:Size_comparisons>. Acesso em 21 de junho de 2016.

[8] JERIN, Mathew. Apple Store growing by over 1000 apps per day. International Business Times. 2014. Disponível em: <http://www.ibtimes.co.uk/apple-app-store-growing-by-over-1000-apps-per-day-1504801>. Acesso em 30 de junho de 2016.

[9] AGILE ALLIANCE. Agile Manifesto. Disponível em: <https://www.agilealliance.org/agile101/the-agile-manifesto/>. Acesso em 30 de junho de 2016.

[10] FIELDING, Roy Thomas. Architectural Styles and the Design of Network-based Software Architectures. 2000. pág. 76. Disponível em: <http://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf>. Acesso em 1 de abril de 2017.

[11] W3C. URI Specification. Disponível em: <https://www.w3.org/Addressing/URL/uri-spec.html>. Acesso em 30 de abril de 2017.

[12] TARASIEWICZ, Phillip; BOHM, Robin. AngularJS. Brainy Software Inc. Primeira Edição, 2014. pág. 8.

[13] ANTONOV, Alex. Spring Boot Cookbook. Packt Publishing. Primeira Edição, 2015.

[14] DOAN, Anhai; RAMAKRISHNAN, Raghu; HALEVY, Alon Y. Crowdsourcing systems on the World-Wide Web. Communications of The ACM. Vol. 54, 86-96, 2011.

APÊNDICE A - Artigo

Rede social para pesquisa e criação de projetos colaborativos

Yulle Pereira Ulguim[1], Frank Augusto Siqueira[1]

[1]Dpt. Informática e Estatística - Universidade Federal de Santa Catarina (UFSC)

Postal 476 - 88040-900 - Florianópolis - SC - Brasil

yulguim@gmail.com, frank.siqueira@ufsc.br

Resumo. *A internet sempre manteve a cultura da colaboração e difusão do conhecimento. Seja com software livre ou com ambientes onde o conteúdo pode ser gerado por qualquer pessoa interessada em colaborar. A cada dia são lançadas milhares de aplicações nas lojas de aplicativos virtuais, grande parte por desenvolvedores independentes. A aplicação desenvolvida neste trabalho tem por objetivo ajudar a estes desenvolvedores e atuantes da área de tecnologia a encontrar colaboradores, sócios e outros profissionais interessados em participar na criação, implementação e execução de suas ideias, além de visar ampliar o alcance e a divulgação da visão colaborativa no mundo da tecnologia. A implementação do sistema foi realizada utilizando tecnologias atuais e padrões de desenvolvimento bastante utilizados no mercado de software atualmente.*

1. Introdução

Crowdsourcing, (em português, Colaboração coletiva), é a aglutinação das palavras *crowd* (grupo de pessoas) e *outsourcing* (terceirização). O termo foi cunhado em 2006 e adicionado no dicionário Merriam-Webster em 2011 com a seguinte definição:

"Crowdsourcing: Prática de obter serviços, ideias ou conteúdo através da contribuição de um grupo de pessoas, especialmente da comunidade virtual; ao invés de métodos tradicionais como empregados e fornecedores". [1]

O trabalho tem como objetivo o desenvolvimento de um sistema que ajude pessoas com interesses em comum a interagirem para criação, desenvolvimento e manutenção de projetos colaborativos, fomentando a cultura de colaboração e a implementação de novas ideias no mercado de software.

1.1 Back-end

A aplicação irá necessitar de um servidor que terá como responsabilidade o tratamento de todas as requisições feitas através da interface do usuário (front-end).

A persistência de dados será realizada utilizando PostgreSQL [2] e o acesso a esta camada, gerenciado pelo Hibernate. A linguagem que será utilizada para desenvolver este lado do software é o Java, juntamente com o framework Spring Boot [3].

1.2 Front-end

O lado do cliente será implementado utilizando HTML e AngularJS [4].

O AngularJS é um framework *front-end* para aplicações web escrito em javascript e desenvolvido e mantido pela Google. Segundo o site Libscore [5], que coleta dados de uso dos principais *frameworks* do mercado, seu uso cresce a cada dia entre os desenvolvedores web. Possui características MVC, *single-page application* e *two-way data binding*, que facilitam e aceleram o processo de desenvolvimento do front-end.

2. Projeto

O sistema proposto será implementado buscando resolver os problemas e situações abordados no capítulo de introdução deste trabalho. O objetivo final é a criação de uma aplicação web que permita que pessoas interessadas em utilizar o sistema possam se cadastrar no mesmo. Os usuários cadastrados poderão manter um perfil pessoal com foto, interesses, experiências em projetos e trabalhos passados, links para portifólio e redes sociais de sites externos, além de endereços de email e telefones de contato, Skype e outras ferramentas de comunicação. Além disso, os usuários poderão também criar e pesquisar páginas sobre projetos que estão sendo desenvolvidos por outras pessoas ou que precisam de voluntários para serem inicializados. Por último, a aplicação contará com uma tela que exibirá projetos relevantes de acordo com o perfil do usuário que está acessando o sistema.

Após implementado, o sistema irá ser hospedado em um servidor dedicado na nuvem. Este servidor irá possuir um servidor de banco de dados PostgreSQL para persistência de dados da aplicação e um servidor *web* Apache para gerenciar as conexões HTTP, permitindo assim a possibilidade de executar múltiplas instâncias da aplicação, balanceando a carga entre todas elas.

A figura 1 mostra a arquitetura do sistema:

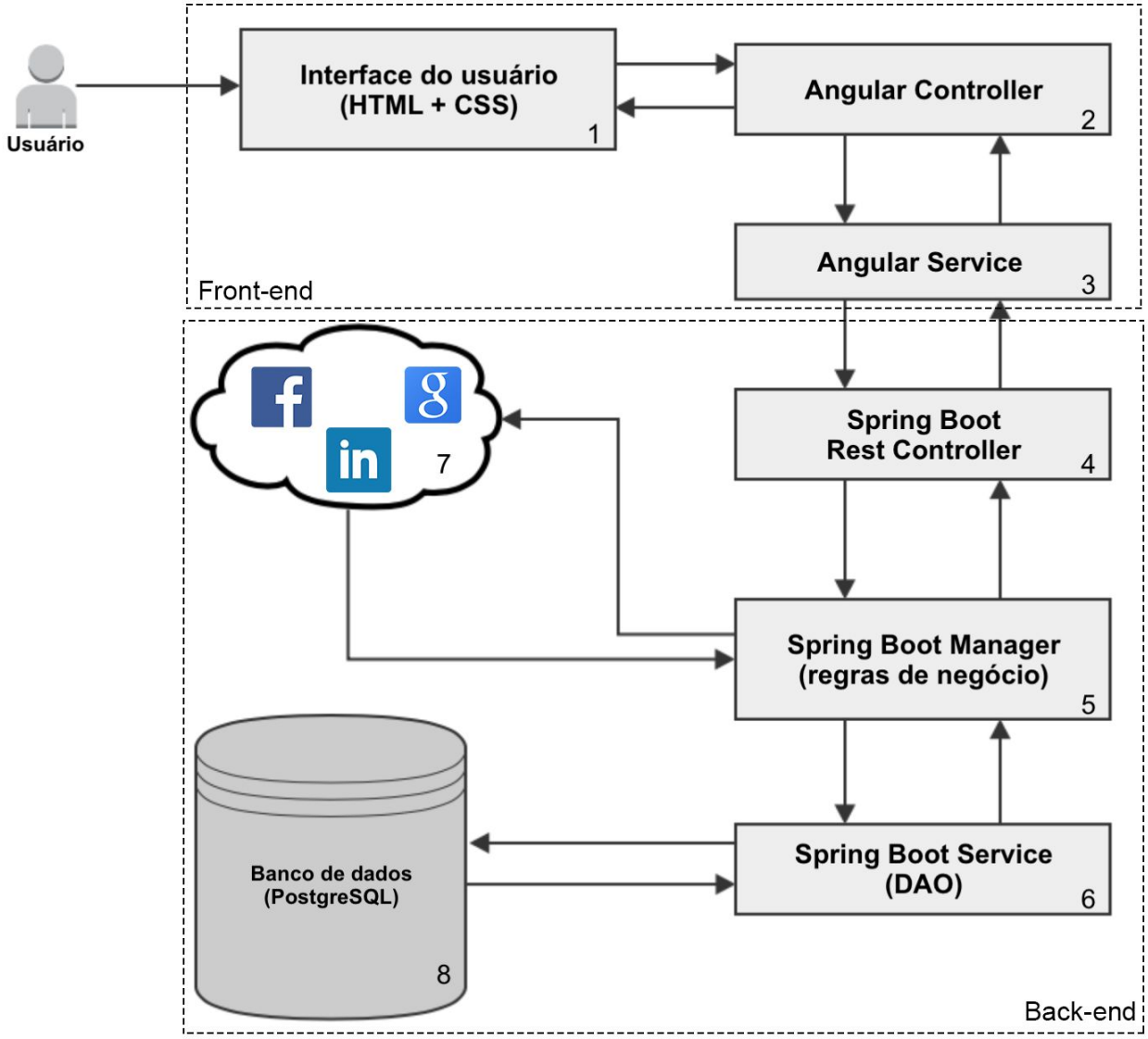


Figura 1 - A arquitetura do sistema

3. Desenvolvimento

É indispensável a implementação de um servidor que irá conter a lógica de negócios, que será responsável por tratar as requisições do lado do cliente. Para isso, utilizaremos a linguagem de programação Java em conjunto com o framework Spring Boot, e o SGBD (Sistema Gerenciador de Banco de Dados) PostgreSQL.

O back-end possui uma organização e hierarquia de arquivos e pacotes bem definida para facilitar o desenvolvimento e manutenção do código escrito. O nome define explicitamente o que está contido em cada pacote. O pacote *controller* armazena todos os controladores de nossa aplicação, o pacote *manager*, todos os *managers* e no *service*

estão todas as classes que tem como objetivo a comunicação com o banco de dados. Isso se repete em toda a aplicação. Cada classe também possui um nome que identifica exatamente o papel que ela tem no programa. Por exemplo, a classe ChatManager tem toda a regra de negócio para gerenciar os *chats* dos usuários. As entidades estão todas contidas no pacote *entity*, e os *DTO's*, que são as entidades utilizadas para expor os dados na tela, no pacote *view*. A figura 2 mostra um exemplo de um método de um *manager*.

```
public ProjetoView save(Profile profile, ProjetoView view) throws ValidationException {
  validate(view);
  Projeto entity;
  if (view.getId() == null) {
     //Cadastrar projeto
     entity = new Projeto();
     entity.setOwner(getAccountLogada(profile));
     entity.setTitulo(view.getTitulo());
     entity.setDescricao(view.getDescricao());
     entity = super.save(entity, profile);

     view.addMessage(new MessageSuccess("success.save"));
  } else {
     //Update
     entity = projetoService.selectById(Projeto.class, view.getId());
     if (!entity.getOwner().getId().equals(getAccountLogada(profile).getId())) {
        throw new ValidationException(new Message("warn.save"));
     }
     entity.setTitulo(view.getTitulo());
     entity.setDescricao(view.getDescricao());
     entity = super.update(entity, profile);

     view.addMessage(new MessageSuccess("success.update"));
  }

  return view;
}
```

Figura 2 - Método *save* do ProjectManager

A interface da aplicação foi desenvolvida utilizando HTML, CSS e o framework AngularJS, desenvolvido pela Google e escrito em JavaScript.

O AngularJS utiliza o recurso de *Single-page Application*, que utiliza uma única página web para exibir a aplicação inteira, promovendo a experiência do usuário e fazendo com que o sistema se pareça com uma aplicação desktop.

Nas SPAs, todo o código principal do programa, como HTML, CSS e JavaScript, é carregado junto da primeira requisição, e outros módulos são carregados dinamicamente conforme necessidade, normalmente em resposta à ações requisitadas pelo usuário. A página nunca é recarregada, apenas as *views* são renderizadas de acordo com a tela que o cliente está querendo acessar.

Assim como no back-end, o front-end da aplicação também está organizado de forma com que os desenvolvedores possam localizar facilmente os arquivos e recursos do programa, tornando simples a manutenção do mesmo. A aplicação na parte do cliente está dividida basicamente em *views*, controladores, serviços e diretivas. A *view* nada mais é do que a página que o usuário está vendo; um trecho de código HTML que define o comportamento e template de uma tela do programa. Toda a interação de uma *view* é gerenciada pelo controlador desta página. Cliques do usuário, carregamento de dados e tudo que envolve ações na página está programada no *controller*. A figura 3 mostra como exemplo o controlador da tela de mensagens.

```
app.controller("mensagensCtrl", ['usuarioLogadoService', "mensagensService", function
(usuarioLogadoService, mensagensService) {
  var vm = this;

  vm.usuarioLogado = {};
  vm.chats = [];
  vm.chatLoaded = {};

  //functions
  vm.loadChat = loadChat;
  vm.saveMensagem = saveMensagem;
  vm.removeMensagem = removeMensagem;

  function loadChat(chat) {
     mensagensService.load(chat.id).success(function(view) {
        console.log(view);
        vm.chatLoaded = view;
     });
  };

  function saveMensagem(mensagem) {
     mensagem.chatId = vm.chatLoaded.id;
     mensagensService.saveMensagem(mensagem).success(function(view) {
           mensagem = undefined;
           vm.chatLoaded.mensagens.push(view);
       });
  };
```

```
   function removeMensagem(mensagem) {
     var index = vm.chatLoaded.mensagens.indexOf(mensagem);
       mensagem.chatId = vm.chatLoaded.id;
       mensagensService.removeMensagem(mensagem).success(function() {
           vm.chatLoaded.mensagens.splice(index, 1);
       });
   };

  var iniciarTela = function() {
     vm.usuarioLogado = usuarioLogadoService.getUsuario();
     mensagensService.list().success(function(data) {
        console.log(data);
        vm.chats = data;
     });
  };

   iniciarTela();
}]);
```

Figura 3 - *mensagensCtrl*

Ao acessar a tela, o método "iniciarTela" é executado, fazendo uma chamada para o *mensagensService*, que por sua vez faz uma requisição do tipo GET para o servidor REST, pedindo todas as conversas do usuário logado no sistema. Esta ação serve para carregar os dados iniciais em tela, ou seja, a lista de mensagens recebidas pelo usuário, de forma com que ele possa carregar um *chat* e responder mensagens.

Todos as funções que iniciam com a expressão *vm* servem para interação com algum elemento da interface. Por exemplo, quando o usuário seleciona uma conversa na lista carregada anteriormente pela função "iniciarTela", o método "loadChat" é invocado, recebendo o *id* do chat que o usuário deseja ver, e passando a requisição para o serviço responsável, que neste caso é o *mensagensService*. A resposta retornada do servidor são as mensagens enviadas e recebidas na conversa selecionada, e os dados são carregados na tela. A figura 4 exibe a tela de mensagens da aplicação.

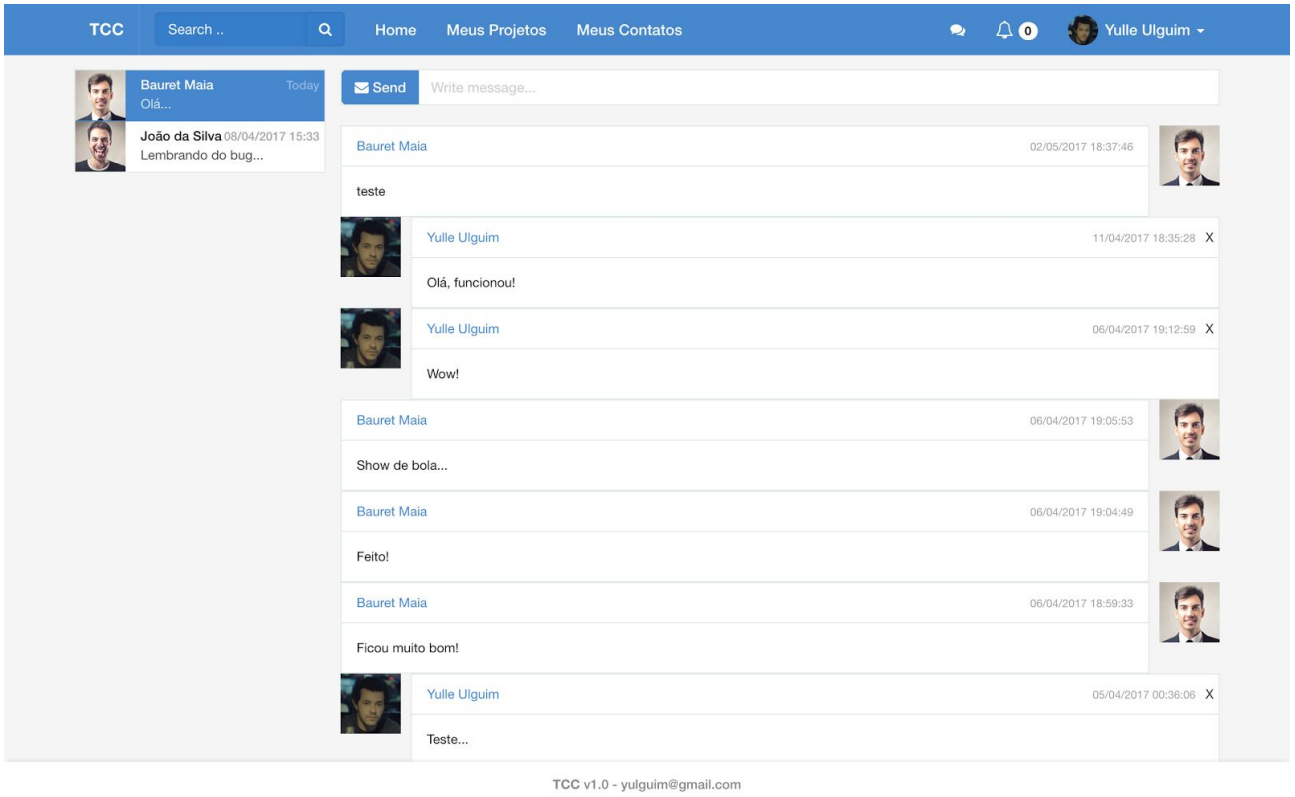


Figura 4 - Interface de mensagens do usuário

4. Conclusão

A motivação para o desenvolvimento desta ferramenta se deu pelo fato de não existir uma opção como esta disponível na internet. Todas as aplicações existentes atualmente têm como objetivo um relacionamento estritamente profissional e de curto prazo entre o dono de um projeto e um profissional *freelancer*, que quase sempre assume um determinado trabalho tendo em vista a sua remuneração.

Espero que este trabalho possa preencher a carência de um nicho pouco explorado, visando fomentar a cultura do compartilhamento de ideias, concentrando pessoas interessadas na inovação e no desenvolvimento de novos aplicativos; e porque não, na criação de novas *startups* e empresas.

Referências

[1] MERRIAM WEBSTER DICTIONARY. Crowdsourcing definition. Disponível em <http://www.merriam-webster.com/dictionary/crowdsourcing>. Acesso em: 04 de junho de 2016.

[2] POSTGRESQL GLOBAL DEVELOPMENT GROUP. PostgreSQL. Disponível em: <http://postgresql.org>. Acesso em: 10 de junho de 2016.

[3] PIVOTAL SOFTWARE. Spring Boot Documentation. Disponível em: <http://projects.spring.io/spring-boot>. Acesso em: 10 de junho de 2016.

[4] ANGULARJS. AngularJS. Disponível em <http://angularjs.org>. Acesso em: 06 de junho de 2016.

[5] LIBSCORE. Libscore. Angular Usage. Disponível em <http://libscore.com/#angular>. Acesso em: 07 de junho de 2017.

APÊNDICE B - Código fonte da aplicação

O código fonte da aplicação desenvolvida neste projeto de conclusão de curso encontra-se inteiramente disponível no repositório *Git* no seguinte endereço:

https://github.com/yulguim/tcc.