Universidade Federal do Estado do Rio de Janeiro
Centro de Ciências Exatas e Tecnologia
Escola de Informática Aplicada

SISTEMA WEB DE ANÚNCIO DE VAGAS EM IMÓVEIS: UM EXEMPLO DE DESENVOLVIMENTO ÁGIL COM RUBY ON RAILS

Matheus Charif Penchel
Rodrigo Cantarela

Orientador
Asterio Kiyoshi Tanaka

RIO DE JANEIRO, RJ – BRASIL.
JUNHO DE 2015

SISTEMA WEB DE ANÚNCIO DE VAGAS EM IMÓVEIS: UM EXEMPLO DE DESENVOLVIMENTO ÁGIL COM RUBY ON RAILS

Matheus Charif Penchel
Rodrigo Cantarela

Projeto de Graduação apresentado à Escola de Informática Aplicada da Universidade Federal do Estado do Rio de Janeiro (UNIRIO) para obtenção do título de Bacharel em Sistemas de Informação.

Aprovada por:

_____

Asterio Kiyoshi Tanaka, Ph.D. (UNIRIO)

_____

Luiz Carlos Montez Monte, D.C. (UNIRIO)

_____

Geiza Maria Hamazaki da Silva, D.C. (UNIRIO)

RIO DE JANEIRO, RJ – BRASIL.
JUNHO DE 2015

AGRADECIMENTOS

Agradecemos aos nossos queridos amigos que nos apoiaram nesta jornada em busca do conhecimento. Aos familiares que sempre estiveram ao nosso lado e a UNIRIO por ser uma universidade de excelência no ensino.

Rodrigo Cantarela e Matheus Charif Penchel

RESUMO

Este trabalho teve como objetivo aplicar conhecimentos adquiridos ao longo do curso de Bacharelado em Sistemas de Informação para desenvolver uma aplicação Web, usando práticas do processo de desenvolvimento ágil Scrum. Foi feita uma breve análise do mercado de locação imobiliária no Estado do Rio de Janeiro, que constatou uma oportunidade de criação de um sistema Web especializado em anúncios de vagas de imóveis. O desenvolvimento do sistema envolveu desde o levantamento de requisitos e modelagem de casos de uso e de classes até a implementação usando o framework Ruby on Rails. A conjugação do uso deste framework com as práticas do Scrum resultou em um desenvolvimento rápido e eficaz do sistema, que pode ser replicado em outras aplicações.

Palavras-chave: Desenvolvimento Ágil, Scrum, Ruby on Rails, Sistema de Aluguel Imobiliário.

ABSTRACT

This work aimed to apply knowledge acquired during the Bachelor in Information Systems course to develop a Web application using practices of the Scrum agile development process. A brief analysis of the real state rental market in the state of Rio de Janeiro was made, which found an opportunity to create a Web system specialized on advertisement of vacancies in properties. The development of the system involved from requirements gathering and modeling of use cases and classes to implementation using the Ruby on Rails framework. The combination of using this framework with Scrum practices resulted in a rapid and efficient development of the system, which can be replicated in other applications.

Keywords: Agile Development, Scrum, Ruby on Rails, Real Estate Rental System.

ÍNDICE

LISTA DE FIGURAS

LISTA DE TABELAS

1 INTRODUÇÃO

1.1 Motivação

Com o grande aquecimento do mercado imobiliário no Estado do Rio de Janeiro e com a realização de um grande evento internacional (Olimpíadas de 2016), existe uma grande oportunidade de demanda para um sistema web de divulgação de aluguel de vagas em imóveis. Essa oportunidade motivou o desenvolvimento de um sistema capaz de oferecer, de forma simples e de fácil acesso, um mecanismo de busca de vagas em imóveis que são cadastrados pelos próprios proprietários e sendo compatível com os principais dispositivos existentes no mercado, tais como *tablets*, *smartphones* e *desktops*.

1.2 Justificativa

O sistema proposto neste trabalho oferece uma solução com software livre e de fácil utilização para os usuários, ajudando-os a buscar e anunciar vagas em imóveis disponíveis no Estado do Rio de Janeiro.

O sistema Imobiliária Web deve possuir uma interface intuitiva e de fácil acesso dando ao usuário opção de buscar vagas em imóveis a partir de suas datas de disponibilidade, preenchendo um formulário de interesse, que será enviado por e-mail ao locador, com todos os dados do interessado. O usuário poderá visualizar as fotos dos cômodos do imóvel e ter acesso a um FAQ onde ele poderá obter ajuda sobre a lei vigente do inquilinato, dicas de cuidados a serem tomados, como se estabelecer um contrato de aluguel e dicas para se estabelecer um ambiente saudável.

Como a ferramenta utilizada para desenvolver o sistema se enquadra perfeitamente em diversos métodos de desenvolvimento ágil, é possível acrescentar funcionalidades ou refatorar o sistema de acordo com novas demandas e/ou observações dos locadores, podendo, no futuro, inclusive, oferecer outras funcionalidades de controle e uso do sistema.

Financeiramente, a sua rentabilidade pode vir de publicidade, da análise e venda de dados em caso de grande procura de locatários, ou até mesmo do sistema ser personalizado para alguma empresa imobiliária.

1.3 Objetivo do projeto

O objetivo deste projeto é consolidar conhecimentos adquiridos através das disciplinas do Bacharelado em Sistemas de Informação (BSI): de desenvolvimento (TP1, TP2, EDD1, EDD2, PCS, PCS-SGBD, PM, PS), de modelagem (FSI, AS), de banco de dados (BD1, BD2), de empreendedorismo (AEA, Empreendedorismo) e de *front-end* (DPW). Adicionalmente, o projeto busca demonstrar a possibilidade de lançar um produto no mercado que respeite as metodologias e técnicas aprendidas no decorrer do curso de BSI.

1.4 Organização do trabalho

O presente trabalho está estruturado em capítulos e, além desta introdução, está organizado da seguinte forma:

Capítulo 2: Análise de Mercado

Neste capítulo, são apresentados a breve análise e o levantamento feito do mercado imobiliário do Estado do Rio de Janeiro, levando em consideração as oportunidades de criação do sistema Imobiliária Web e uma possível forma de capitalização do sistema.

Capítulo 3: Levantamento de Requisitos e Modelagem do Sistema

São descritos os requisitos funcionais e não funcionais do sistema, dos principais casos de uso e do diagrama de classes resultante do banco de dados do sistema.

Capítulo 4: Metodologia e Desenvolvimento Ágil

São abordados os tópicos referentes ao uso de metodologia ágil em desenvolvimento de software, uma breve descrição do Scrum e de como ele será utilizado no desenvolvimento do sistema Imobiliária Web junto com a descrição dos principais *sprints* de desenvolvimento.

Capítulo 5: Implementação usando Ruby on Rails

A tecnologia utilizada no projeto e a construção do sistema são mostradas neste capítulo, com suas peculiaridades e com uma descrição detalhada das bibliotecas adotadas para o desenvolvimento. Toda a infraestrutura utilizada é também explicitada e justificada neste capítulo.

Capítulo 6: Conclusão

O capítulo descreve o resultado final do trabalho, evidenciando a facilidade do uso da ferramenta pelos usuários finais, os pontos fortes e fracos de todo o processo adotado, contribuições, limitações e trabalhos futuros.

2 ANÁLISE DE MERCADO

Este capítulo relata uma breve análise informal de mercado que, através de consultas a páginas especializadas e buscas na Internet, justifica a relevância do sistema desenvolvido junto ao mercado imobiliário.

2.1 Índice FIPE-ZAP

O mercado imobiliário encontra-se muito aquecido no Brasil, principalmente no Rio de Janeiro, o que aumenta a motivação para desenvolver um projeto nessa área. O índice FIPE-ZAP é fornecido por duas instituições conforme a descrição abaixo:

> "FIPE, Fundação Instituto de Pesquisas Econômicas, é uma fundação de direito privado, sem fins lucrativos, criada em 1973, que apoia o Departamento de Economia da Faculdade de Economia, Administração e Contabilidade da Universidade de São Paulo (FEA-USP) nas áreas de pesquisa e ensino.
> O ZAP, uma empresa do Globo, é o mais completo, moderno e eficiente portal de classificados da Internet brasileira " [Globo, 2015]

Em uma consulta realizada em Janeiro de 2015, o gráfico abaixo demonstra a variação do índice FIPE-ZAP sobre transações de aluguel durante o período 2008-2014 na cidade do Rio de Janeiro.

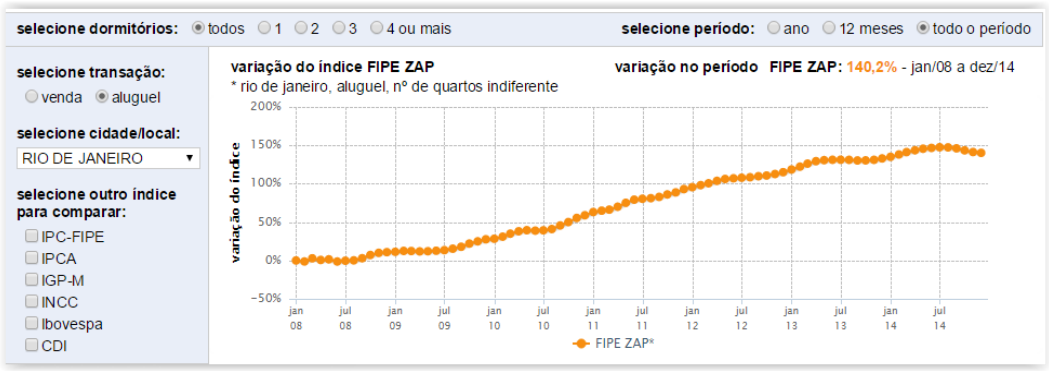

Figura 1 - Índice FIPE-ZAP
fonte: http://www.zap.com.br/imoveis/fipe-zap-b/

É possível observar a valorização dos aluguéis após o anúncio do Brasil como sede da Copa do Mundo de 2014, o que aconteceu no final de 2007. Até a realização da Copa do Mundo, o gráfico indica uma valorização que chega a 150%

no início do evento; após a realização deste evento, uma pequena desvalorização ocorre. Com a realização das Olimpíadas de 2016 na cidade do Rio de Janeiro, há um possível crescimento na valorização do mercado imobiliário, indicando assim uma oportunidade para o desenvolvimento do sistema Imobiliária Web.

2.2 Levantamento da situação atual

Conforme pesquisa feita em mecanismos de busca, não existe um sistema gratuito e sem burocracias para o anúncio de vagas em imóveis por locadores. Para que um locador anuncie seu imóvel, ou paga uma taxa para a utilização do sistema ou paga um valor para o sistema ou pessoa responsável por uma indicação. Todos os sistemas encontrados tinham foco somente em anúncios de vagas de imóveis inteiros.

2.3 Sobre a Lei do Inquilinato

Embora não haja uma pesquisa sobre esse tema, supõe-se que pessoas interessadas em alugar vagas em imóveis, assim como locadores de vagas, não possuam o perfil usual de locadores e locatários de imóveis. Assim, um sistema para atender a esse público deve oferecer dicas aos seus usuários sobre as práticas e cuidados a serem tomados ao se estabelecer um contrato de aluguel, com base na Lei do Inquilinato (Lei nº 8.245, de 18 de outubro de 1991). Essa Lei, que trata das obrigações e direitos tanto do locatário quanto do locador, deve estar disponível e atualizada no sistema para consultas e esclarecimentos.

2.4 Solução esperada

De acordo com o levantamento da situação atual, é desejável uma solução de fácil acesso e extremamente prática, seguindo assim os requisitos levantados. Trata-se de um sistema *web*, gratuito, que possua funcionalidades que facilitam o anúncio de vagas por parte do locador e também a busca de vagas por parte de possíveis locatários.

O sistema deve permitir que um locador cadastre uma conta, sem qualquer custo, podendo assim cadastrar seus imóveis. Feito isto, deve ser possível cadastrar seus quartos, classificando-os de acordo com o tipo e adicionando os móveis nele contidos.

Por parte de um possível locatário, basta que procure uma vaga de seu agrado, preencha um formulário de reserva e aguarde o contato do locador, já que um e-mail com o formulário será enviado ao administrador do sistema e este fara o contato notificando o interesse da vaga. Caso não encontre o que procura, pode preencher um formulário com as informações da vaga que procura e o administrador ao receber a notificação irá avisar a todos os locadores que tenham o perfil da vaga desejada. O locatário após ser notificado pode entrar em contato com o interessado, para verificar se ainda a interesse e estabelecer o contrato de aluguel.

Em caso de vaga preenchida por um locatário, basta o locador alterar a data de disponibilidade desta, para que outros potenciais locatários não preencham os formulários de vagas ocupadas. Toda a parte jurídica e financeira deve ser feita fora do sistema de aluguéis de vagas.

2.5 Possibilidades de arrecadação financeira

A arrecadação financeira no modelo de negócio do sistema depende única e exclusivamente da forte adesão de locadores e locatários, de forma que seja possível publicar propagandas no sistema. Além disso, o sistema pode ser personalizado para alguma imobiliária específica, facilitando assim seus controles de vagas e também modernizando sua publicidade.

Em caso de grande procura por algum tipo específico de vaga, esta informação poderia ser negociada com grandes imobiliárias para lhes dar uma vantagem no mercado imobiliário.

3 LEVANTAMENTO DE REQUISITOS E MODELAGEM DO SISTEMA

Este capítulo descreve o levantamento dos principais requisitos funcionais e não funcionais, os principais casos de uso e o diagrama de classe referente ao Banco de Dados utilizado para construção do sistema Imobiliária Web, aplicando as melhores práticas de documentação.

3.1 Levantamento de requisitos

Abaixo seguem os requisitos funcionais; Requisitos que são traduzidos em funcionalidades dentro do sistema; na seção seguinte, são apresentados os requisitos não-funcionais, ou seja, os requisitos que independem da implementação do sistema, mas que são necessários para o seu uso.

3.1.1 Requisitos Funcionais

RF01: O sistema deve permitir que um usuário locador cadastre e altere seus dados cadastrais.

RF02: O sistema deve permitir que um usuário locador cadastre o imóvel, altere dados do imóvel ou exclua um imóvel.

RF03: O sistema deve permitir que um usuário locador cadastre um quarto do imóvel cadastrado, altere os dados e exclua o quarto.

RF04: O sistema deve permitir que um usuário locatário busque uma vaga.

RF05: O sistema deve permitir que um usuário locatário preencha um formulário declarando interesse em uma vaga.

RF06: O sistema deve conter um formulário que deve ser preenchido por um locatário caso ele mostre interesse por uma vaga.

RF07: O sistema deve permitir que um usuário tenha acesso à Lei n° 8.245 em formato digital.

RF08: O sistema deve permitir que o usuário administrador preencha as mensagens de aviso e demais textos do sistema.

RF09: O sistema deve disponibilizar uma página de FAQ, com dicas para os locatários sobre boas práticas de convívio em quartos dentro de imóveis e dicas de como controlar despesas fixas e variáveis.

3.1.2 Requisitos não funcionais

RNF01: O sistema deve ser desenvolvido para a web.

RNF02: O sistema deve poder ser acessado independentemente do dispositivo.

RNF03: O sistema deve ter um design responsivo.

RNF04: O sistema deve ser desenvolvido em Ruby on Rails.

RNF05: O sistema deve utilizar o banco de dados PostgreSQL.

RNF06: O sistema deve ter uma interface de usuários que seja agradável e intuitiva.

3.2 Diagrama de Casos de Uso

Abaixo seguem os diagramas dos principais casos de uso do sistema Imobiliária Web:

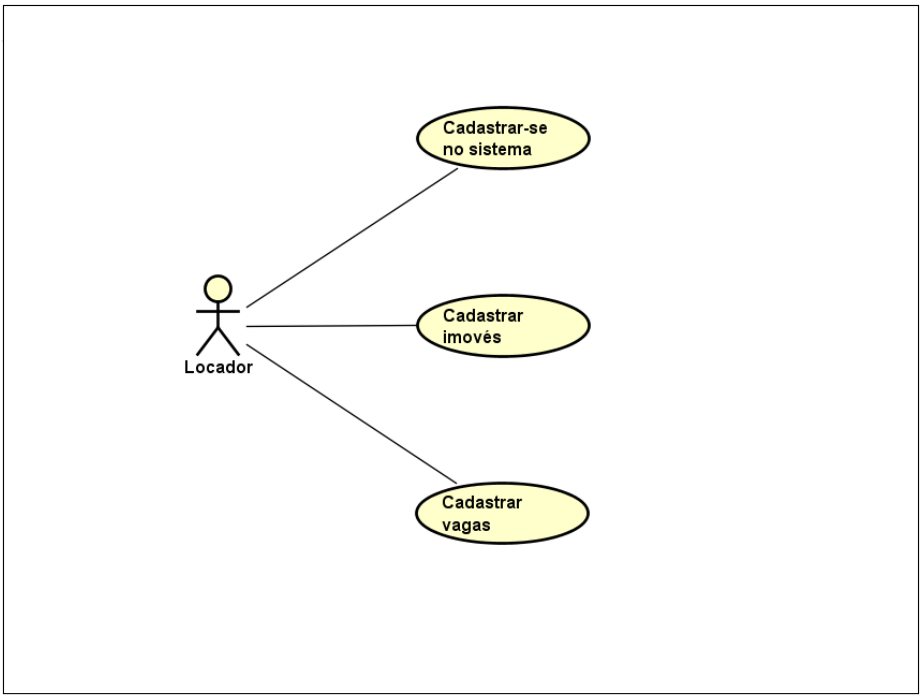

Figura 2 - Caso de uso – Cadastrar

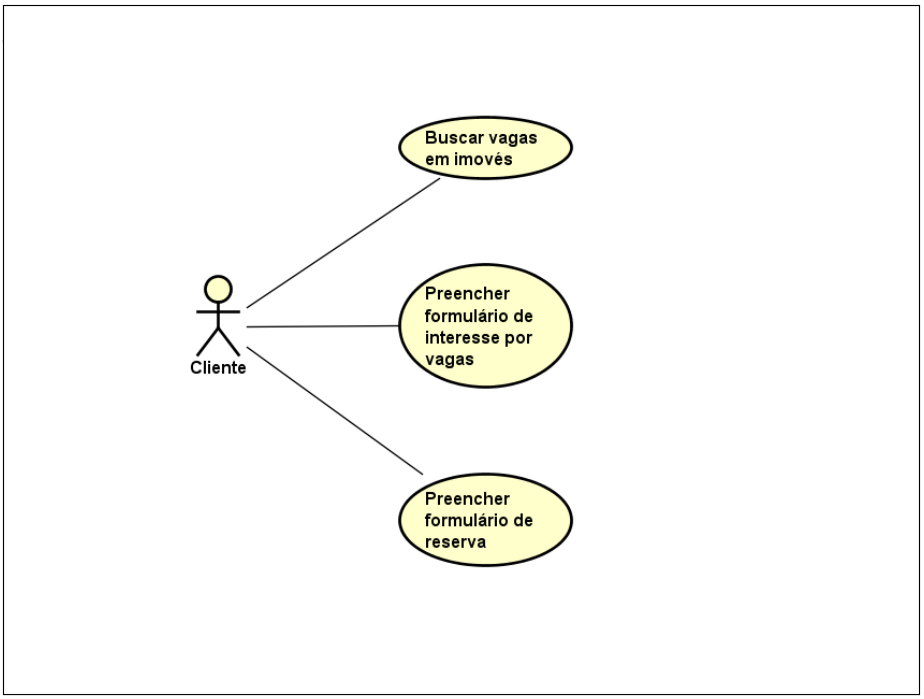

Figura 3 - Caso de uso – Preencher e Buscar

3.3 Casos de Uso

Seguem as descrições dos principais casos de uso do sistema, ilustrados nos diagramas.

3.3.1 Cadastrar

3.3.1.1 Cadastrar-se no sistema

Descrição sucinta:

O sistema irá permitir que um usuário locador se cadastre no sistema de forma gratuita e rápida.

Atores:

Locador.

Requisitos:

RF01

Pré-condições:

Usuário ter um email.

Pós-condições:

Um email de confirmação é emitido para o usuário locador para verificação de conta.

Campos:

e-mail, senha

Fluxo principal:

A. O usuário irá clicar em cadastrar conta e preencher os campos solicitados.

Fluxo alternativo:

N/A.

Fluxo de exceção:

E1 - O sistema irá emitir uma mensagem de erro caso um email inválido seja cadastrado.

E2 - O sistema irá emitir uma mensagem de erro caso usuário digite uma confirmação de senha inválida e direcionado de volta ao campo senha.

3.3.1.2 Cadastrar imóveis

Descrição sucinta:

O sistema permite que o usuário cadastre dados do imóvel do qual pretende divulgar vagas de quartos disponíveis para aluguel.

Atores:

Locador.

Requisitos:

RF02.

Pré-condições:

O usuário locador deve possuir uma conta cadastrada no sistema.

Pós-condições:

O imóvel cadastrado aparecerá nas opções de apartamento ao cadastrar um quarto.

Campos:

Tamanho em m², quantidade de banheiros, quantidade de salas, quantidade de áreas de serviço, cidade, rua, bairro, número e apartamento

Fluxo principal:

A. Usuário deve clicar em Registre-se no cabeçalho do site e preencher os dados do formulário.

Fluxo alternativo:

N/A

Fluxo de exceção:

E1 - O sistema irá emitir erro caso o usuário coloque algum dado no formulário que não esteja de acordo com as regras definidas de cada campo.

3.3.1.3 Cadastrar quarto

Descrição sucinta:

O sistema permite que o usuário locador seja capaz de cadastrar os quartos disponíveis para aluguel em cada imóvel cadastrado.

Atores:

Locador.

Requisitos:

RF03.

Pré-condições:

O usuário deve ter cadastrado um imóvel no sistema.

Pós-condições:

O quarto cadastrado será publicado no site como quarto disponível para alugar.

Campos:

Descrição, data de disponibilidade, apartamento, preço, tipo de quarto, código, tamanho, móveis e foto principal.

Fluxo principal:

A. O usuário ao entrar na listagem de quartos clica em cadastrar quarto.
B. O usuário após clicar cadastrar quarto, preenche o formulário com os campos pedidos e clica em salvar.

Fluxo alternativo:

N/A

Fluxo de exceção:

E1 - Caso o usuário não cadastre algum campo obrigatório do formulário o sistema irá emitir uma mensagem de erro.

E2 - Caso o usuário de preencha algum campo com algum caractere não permitido ele irá emitir uma mensagem de erro.

3.3.2 Preencher

3.3.2.1 Preencher formulário reserva

Descrição sucinta:

O usuário cliente poderá preencher um formulário onde ele demonstra interesse por uma vaga, solicitando reserva por um período determinado de tempo.

Atores:

Cliente.

Requisitos:

RF05.

Pré-condições:

N/A

Pós-condições:

Um email irá ser enviado ao locatário com os dados do interessado pela vaga.

Campos:

Nome completo, e-mail, telefone, data de nascimento, check-in, check-out e cpf.

Fluxo principal:

A. O usuário ao fazer a busca de quartos disponíveis
B. Clicar no quarto desejado
C. Selecionar data de check-in e check-out
D. Clicar em Reserve agora.

Fluxo alternativo:

N/A.

Fluxo de exceção:

E1 - O sistema irá emitir uma mensagem de erro caso algum campo tenha sido deixado em branco ou preenchido com algum caractere não permitido.

3.3.2.2 Preencher formulário de interesse por vaga

Descrição sucinta:

O usuário cliente poderá preencher um formulário de interesse por vaga para ser avisado quando alguma estiver disponível.

Atores:

Cliente.

Requisitos:

RF06.

Pré-condições:

N/A

Pós-condições:

O usuário cliente será avisado por email quando a vaga desejada estiver disponível no sistema.

Campos:

Nome completo, e-mail, telefone, data de nascimento, check-in, check-out, cpf, em que cidade gostaria de ficar, valor mínimo e valor máximo.

Fluxo principal:

A. O usuário deve clicar no menu “não consegue encontrar um quarto”.
B. O usuário deve preencher os campos do formulário.

Fluxo alternativo:

N/A.

Fluxo de exceção:

E1 - O sistema irá emitir uma mensagem de erro caso algum campo tenha sido deixado em branco ou preenchido com algum caractere não permitido.

3.3.3 Buscar

3.3.3.1 Buscar vagas de quartos em imóveis.

Descrição sucinta:

O usuário cliente poderá realizar busca de vagas disponíveis nos imóveis cadastrados no sistema.

Atores:

Cliente.

Requisitos:

RF04.

Pré-condições:

N/A

Pós-condições:

O usuário cliente irá visualizar todas as vagas disponíveis no sistema referente aos critérios de busca selecionados.

Campos:

Mês, ano, estado e Bairro

Fluxo principal:

A. O usuário deve selecionar os critérios de busca que ele deseja para a busca de vaga.
B. O usuário deve clicar no botão busca.

Fluxo alternativo:

N/A.

Fluxo de exceção:

E1 - O sistema irá emitir uma mensagem de erro caso algum dos campos não forem preenchidos.

E2 – O sistema irá emitir uma mensagem de erro caso nenhuma vaga seja encontrada com os critérios definidos pelo usuário.

3.4 Diagrama de Classes

Abaixo segue o diagrama de classes gerado a partir de uma biblioteca para Ruby on Rails chamada RailRoady [Smaldone, 2008] que gera alguns diagramas UML. As classes estão explicadas a seguir.

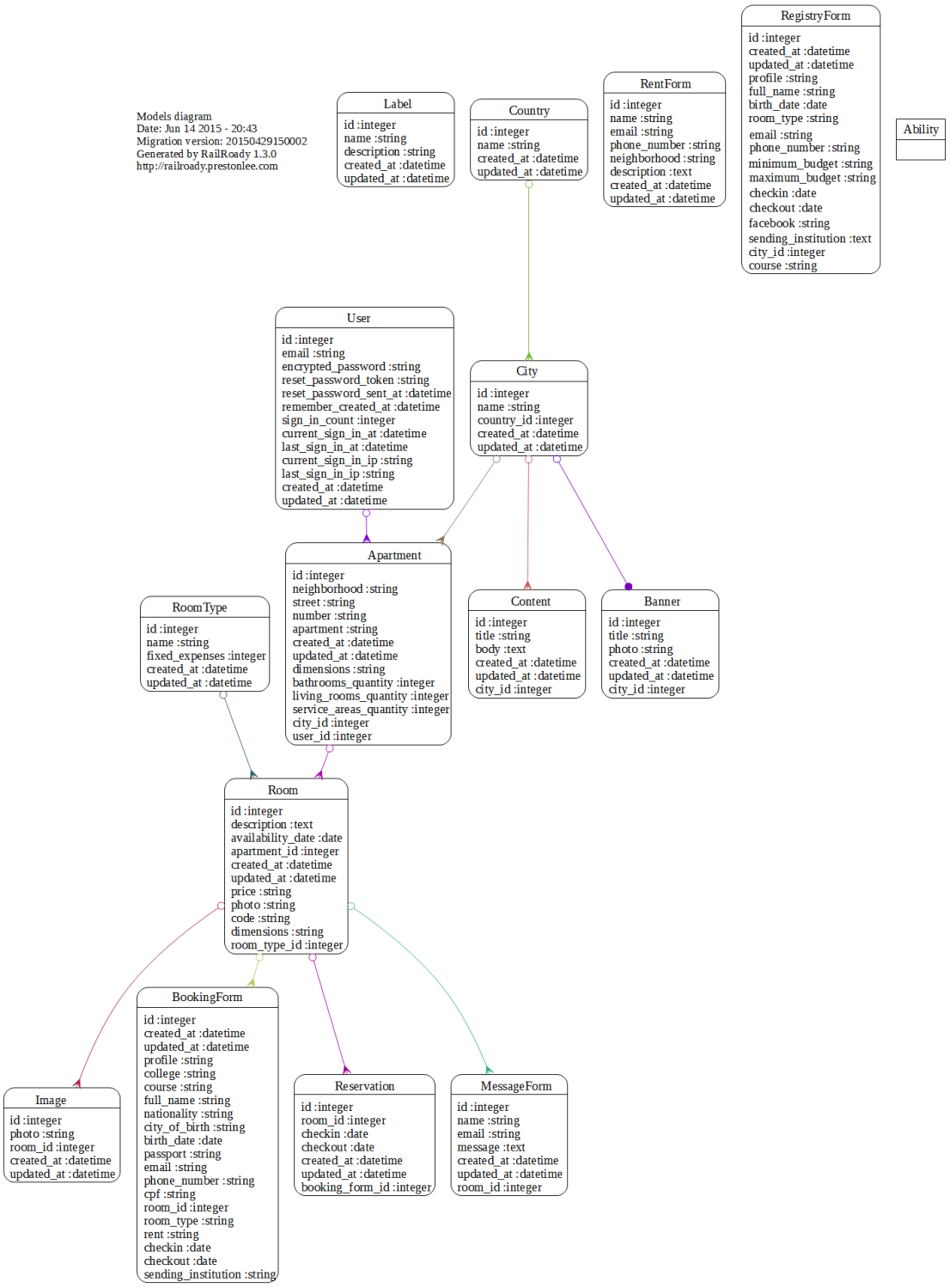

Figura 4 - Diagrama de Classe

A classe User é utilizada para cadastrar e realizar o *login* de locadores, que por sua vez podem possuir inúmeros Apartments e, dentro destes, diversos Rooms, que possuem RoomTypes. A classe Image armazena as fotos destes Rooms, enquanto o BookingForm é o formulário, que fica armazenado no sistema, para guardar o interesse de possíveis locatários em relação a algum quarto específico. Uma vez que um Room é formalmente alugado, o User associado pode registrar uma Reservation, com datas de *check-in* e *check-out* para que o sistema entenda quando este quarto estará novamente disponível.

O MessageForm é um formulário de mensagem que está disponível na página pública de cada quarto, servindo para notificar o User associado de qualquer dúvida, observação ou crítica apresentada por algum visitante. Um Apartment está ligado a uma City que por sua vez está ligado a um Country. A classe Content serve para criar páginas estáticas para cada cidade cadastrada no sistema, caso o administrador ache interessante tê-las; cada cidade possui também um Banner, que é mostrado na *home* do sistema.

Os Labels são mensagens e rótulos que estão espalhados pelo sistema e que podem ser modificados a qualquer momento pelo administrador. Tanto o RentForm quanto o RegistryForm são formulários, onde o primeiro é uma tabela que permite que um possível locador entre em contato com a administração do *website*, enquanto o segundo é um formulário que permite que possíveis locatários descrevam os quartos que estão procurando caso não o encontrem no sistema.

Por último, a tabela Ability é responsável por armazenar as informações de permissão de usuários.

A figura 5 apresenta as tabelas do banco de dados do sistema Imobiliária Web, no PostgreSQL, visualizadas através da ferramenta PGAdmin III.

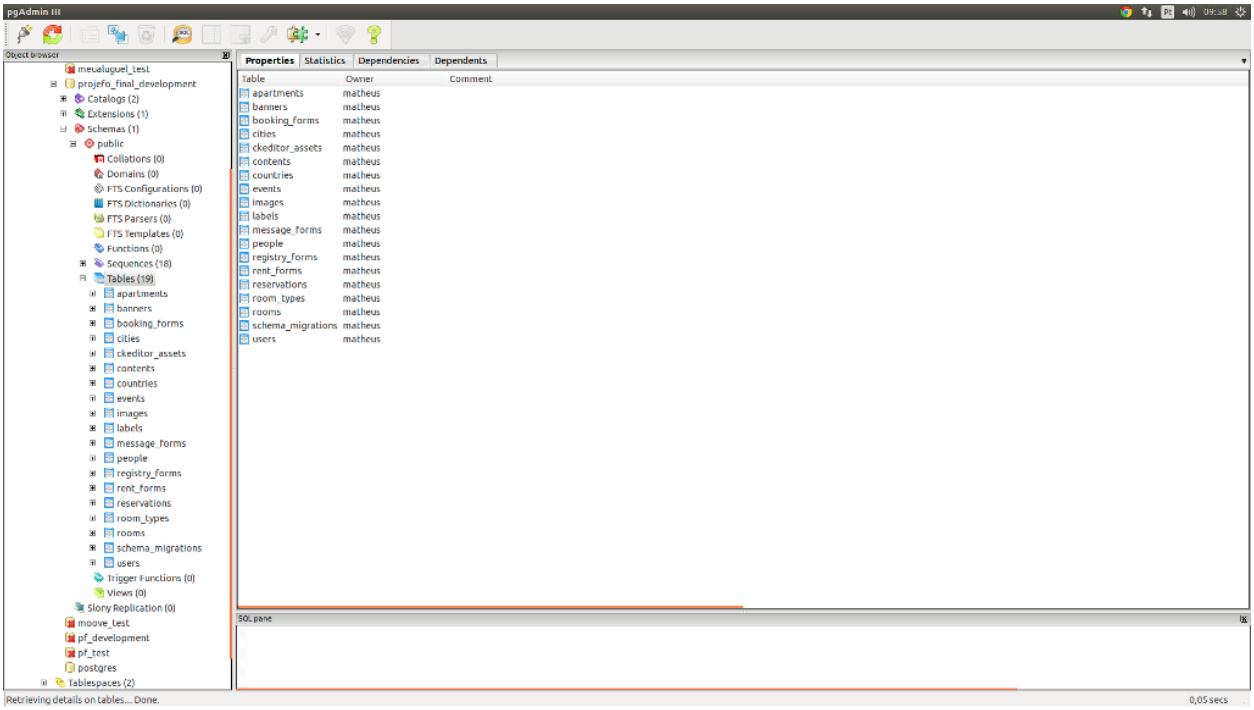

Figura 5 - Tabelas no PostgreSQL

4 METODOLOGIA E DESENVOLVIMENTO ÁGIL

Depois de realizada a análise de mercado e o levantamento de requisitos descritos nos capítulos precedentes, neste capítulo abordamos a metodologia de desenvolvimento ágil que utilizada na implementação do sistema Imobiliária Web. Será apresentada uma breve introdução histórica da metodologia e sua vantagem na aplicação deste projeto.

4.1 Antecedentes

A expressão “Metodologias Ágeis” revelou-se mundialmente em 2001, por meio de uma reunião entre 17 especialistas em processos de desenvolvimento de software, onde foi criada a Aliança Ágil e, em consequência, surgiu o tão conhecido “Manifesto Ágil” [Kent, 2001].

São estes os signatários do Manifesto Ágil: Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland e Dave Thomas.

O Manifesto para o Desenvolvimento Ágil de Software, como foi definido, estabelece alguns princípios e conceitos, como:

- Pessoas e interações, ao contrário de processos e ferramentas.
- Software executável, ao contrário de documentação extensa e confusa.
- Colaboração do cliente, ao contrário de constantes negociações de contratos.
- Respostas rápidas para as mudanças, ao contrário de seguir planos previamente definidos.

4.2 O Manifesto Ágil

Segundo o Manifesto Ágil, a maior prioridade é satisfazer o cliente através da entrega contínua e adiantada de software com valor agregado. Ao contrário do que a filosofia clássica de desenvolvimento de software sugere, as mudanças nos requisitos são bem-vindas, mesmo que tardiamente no desenvolvimento. Isto porque processos ágeis tiram proveito das mudanças visando vantagem competitiva para o cliente.

Nele consta também a ideia de entrega contínua, pois entregar frequentemente software funcionando, de algumas semanas a poucos meses, com preferência à menor escala de tempo, ajuda o cliente a verificar se suas demandas estão sendo atendidas. Para isso, pessoas de negócio e desenvolvedores devem trabalhar diariamente em conjunto por todo o projeto.

É fundamental que os projetos sejam construídos ao redor de indivíduos motivados que precisam de todo ambiente e suporte necessário para que se sintam confortáveis; é importante ressaltar que, para que isso aconteça, é preciso confiar em suas capacidades.

A ideia é que a medida primária de progresso do projeto seja software funcionando, e para isso é necessário um desenvolvimento sustentável, ou seja, que os patrocinadores, desenvolvedores e usuários sejam capazes de manter um ritmo constante indefinidamente.

O método mais eficiente e eficaz de transmitir informações para e entre uma equipe de desenvolvimento é através de conversa face a face: deve-se cortar reuniões prolongadas e principalmente demandas via e-mail ou alguma outra ferramenta semelhante. Uma prática muito utilizada por quem segue o *Extreme Programming* e que atende a essa ideia, por exemplo, é o *Stand-up Meeting*, que deve ocorrer em poucos minutos e tem como objetivo manter toda a equipe atualizada sobre o que cada um está fazendo, que problemas está encontrando e qual será sua próxima tarefa.

Se software funcionando é a medida primária de progresso, deve-se ter contínua atenção à excelência técnica e bom design, para aumentar a agilidade do processo, desta forma simplificando toda e qualquer refatoração de código que possa vir de novos requisitos.

É também importante ter simplicidade, visto que as melhores arquiteturas, requisitos e designs emergem de equipes auto organizáveis, voltando ao ponto de ter confiança e dar liberdade aos profissionais envolvidos no projeto.

Para melhorar a auto-organização da equipe, é sugerido que, em intervalos regulares, a equipe reflita sobre como se tornar mais eficaz para então refinar e ajustar seu comportamento de acordo.

4.3 Desenvolvimento ágil com Ruby on Rails

Segundo definição do site [Hansson, 2003] *"Ruby on Rails é um framework de desenvolvimento web (gratuito e de código aberto) otimizado para a produtividade sustentável e a diversão do programador*". Ele procura tornar mais fácil o desenvolvimento e instalação dos sistemas web. É focado em produzir sistemas centrados no banco de dados, isto é, seu principal trabalho é manipular dados sem que os utilizadores do sistema necessitem de utilizar a linguagem SQL. Para que isso possa ser efetuado, o Rails utiliza o ActionPack, uma biblioteca para auxílio na geração das páginas que se comunicam com o banco de dados.

Neste projeto, o sistema Imobiliária Web foi desenvolvido utilizando o framework Ruby on Rails e obedecendo à arquitetura MVC (Model View Controller), separando cada parte do código em sua camada, de modo que estas partes interagem de uma forma única.

4.3.1 Scrum

Segundo definição de seus idealizadores [Schwaber, 2013], Scrum é "*um framework para desenvolvimento e manutenção de produtos complexos*". Os projetos são divididos em ciclos (tipicamente mensais) chamados de *Sprints*. O *Sprint* representa um *Time Box* dentro do qual um conjunto de atividades deve ser executado. Metodologias ágeis de desenvolvimento de software são iterativas e incrementais, ou seja, o trabalho é dividido em iterações incrementais, que são chamadas de *Sprints* no caso do Scrum. A imagem abaixo exemplifica o ciclo de desenvolvimento do Scrum.

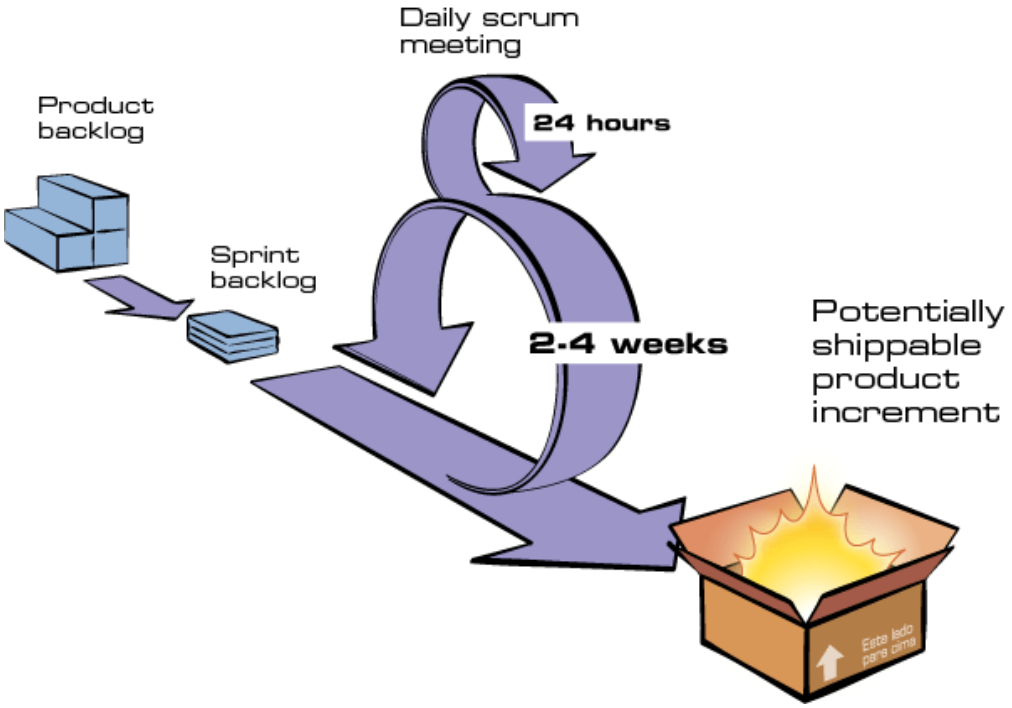

Figura 6 - Ciclo de desenvolvimento Scrum
fonte: http://www.desenvolvimentoagil.com.br/scrum/

4.3.2 Sprints

No Scrum, o trabalho é realizado em iterações ou ciclos de até um mês. O trabalho realizado em cada *sprint* deve criar algo de valor tangível para o cliente ou usuário. Sprints são *timeboxed* (isto é, possuem duração fixa) para que tenham sempre um início e fim (datas fixas), e, idealmente, todos eles devem estar com a mesma duração.

O *Sprint Backlog* é uma lista de tarefas que o *Scrum Team* se compromete a fazer em um *Sprint* como um potencial incremento de produto entregável. Os itens do *Sprint Backlog* são extraídos do *Product Backlog* pela equipe, com base nas prioridades definidas pelo *Product Owne*r e a percepção da equipe sobre o tempo que será necessário para completar as várias funcionalidades; tal noção de tempo está explicitada a seguir. A quantidade de itens do *Product Backlog* que serão trazidos para o *Sprint Backlog* é definida pelo *Scrum Team* que se compromete a implementá-los durante o *Sprint*. Os itens do *Product Backlog* são extraídos a partir de *User Stories*, que são quebradas e divididas em tarefas até que qualquer membro da equipe de desenvolvimento seja capaz de entendê-las e implementá-las, desta forma estando prontas para entrar em um *Sprint Backlog*.

4.3.3 Scrum points

Após a divisão das *User Stories* em tarefas, é prática comum no framework Scrum pontuar as tarefas. Diversos conjuntos de pontos são utilizados no mercado de trabalho, geralmente se baseando na sequência de Fibonacci, com algumas modificações; o ½ é usado ao invés do número 2 pois os *decks* de *poker eram* comercializados com esta carta. O conjunto de pontos adotado neste projeto é: ½, 1, 3, 5, 8, 13.

Segue a idéia geral de cada um desses pontos:

- ½: Tarefa de baixa complexidade e rapidamente executada.
- 1: Tarefa de baixa complexidade mas que exige algum tempo.
- 3: Tarefa de média complexidade ou que exija um tempo considerável.
- 5: Tarefa de média complexidade mas que exige bastante tempo.
- 8: Tarefa de alta complexidade ou que exija muito tempo.
- 13: Tarefa de alta complexidade, onde os desenvolvedores não têm muita ideia de como executá-la ou de quanto tempo irá levar. Costuma significar que a tarefa deve ser dividida em sub-tarefas.

4.3.4 Product Backlog

Abaixo segue o *Product Backlog* para o sistema Imobiliária Web, com as histórias dos Usuários divididas em Tarefas e com seus respectivos pontos.

User Story	Tarefas	Pontos
O locador deve ser capaz de cadastrar um apartamento com todas as vagas disponíveis dentro dele, informando móveis existentes, preço e cadastrando fotos.	CRUD[1] de Apartamentos.	3
	CRUD de Quartos.	3
	Conexão com o S3 da Amazon para armazenar fotos de Quartos.	3
	CRUD de Móveis.	3
O possível locatário deve ser	Filtro de quartos por cidade, mês e	5

[1] Para melhor entendimento CRUD é o acrônimo de Create, Read, Update, Delete

capaz de encontrar um quarto que lhe interesse, de acordo com a cidade, o mês e o ano em que deseja ocupar a vaga. Quando encontrar o que procura, deve poder avisar ao locador o seu interesse, preenchendo um formulário com todos os dados relevantes.	ano.	
	Formulário de declaração de interesse em um quarto específico.	1
	Funcionalidade que altera a disponibilidade do quarto uma vez que ele é ocupado.	1
	Funcionalidade de envio de e-mail.	3
	Formulário de declaração de interesse por quarto enviado por e-mail ao locador.	1
Os possíveis locatários devem encontrar as vagas mais próximas na página principal do sistema, assim como preencher um formulário informando o seu interesse caso não tenha encontrado uma vaga que lhe interesse.	Desenvolver a index da página principal.	5
	Disponibilizar quartos livres mais recentes na página principal.	3
	Formulário de aviso de interesse quando não encontra um quarto que lhe interessa.	1
	Desenvolver menu.	1
O possível locatário deve ter a Lei do Inquilinato disponível a todo o momento, para que ele saiba exatamente o que deve e não deve fazer quando ocupar uma vaga.	Lei do Inquilinato disponível em uma aba do sistema.	1
Um possível locador deve ser capaz de criar uma conta de	Desenvolver classe de usuários.	3
	Desenvolver permissões de usuários.	1

maneira fácil e rápida.	Permitir o *sign up* de qualquer locador.	1
O administrador do sistema deverá ter a opção de modificar os textos do sistema para melhor adequar os anúncios de acordo com a época em que se encontra.	Implementar permissão de administrador.	1
	Renderizar os alertas nas *views* corretas.	1
	CRUD de alertas.	3
O sistema deve estar no ar, com um layout agradável e com as informações sendo persistidas em um banco de dados.	Fazer o *deploy* do sistema no Heroku.	3
	Incorporar as bibliotecas do Twitter Bootstrap.	5
	Configurar o banco de dados PostgreSQL.	1
	Fazer o design das páginas do sistema.	8
Total de pontos: 61		

Tabela 1 - Product Backlog

4.3.5 Sprints

Abaixo segue a divisão de Sprints adotada no desenvolvimento do sistema:

Sprint 1 (12 pontos) - Janeiro: Desenvolver a Index (5), Configurar o banco de dados PostgreSQL (1), Fazer o *deploy* do sistema no Heroku (3), Desenvolver a classe de Usuários (3).

Motivação: A equipe achou interessante desenvolver primeiro a interface do sistema, assim como preparar boa parte da sua infraestrutura. Ao criar o projeto, o banco de dados automaticamente fica pronto, facilitando assim a criação da classe e tabela de Usuários. Feito isto, foi hospedado o projeto no Heroku, deixando pronto assim o *deploy*.

Sprint 2 (13 pontos) - Fevereiro: Fazer o design das páginas do sistema (8), Incorporar as bibliotecas do Twitter Bootstrap (5).

Motivação: Como a index tem um layout único, pouca coisa dela pode ser aproveitada para as demais páginas. Foi priorizado o design de todas as páginas pensadas do sistema, assim como a incorporação das bibliotecas do Twitter Bootstrap, que facilita a implementação deste design.

Sprint 3 (12 pontos) - Março: CRUD de Apartamentos (3), CRUD de Quartos (3), CRUD de Móveis (3), CRUD de Alertas (3).

Motivação: Com toda a parte visual pronta e com a infraestrutura praticamente terminada, chega a hora da equipe dar atenção única e exclusivamente ao desenvolvimento. Os CRUDs de Apartamentos, Quartos, Móveis e Alertas para já podermos "*mockar"* boa parte do sistema.

Sprint 4 (12 pontos) - Abril: Filtro de quartos por cidade, mês e ano (5), Formulário de declaração de interesse em um quarto específico (1), Funcionalidade que altera a data do quarto uma vez que ele é ocupado (1), Formulário de declaração de interesse por quarto enviado por e-mail ao locador (1), Funcionalidade de envio de e-mail (3), Desenvolver permissões de usuários (1).

Motivação: Com quase tudo desenvolvido para os locadores, o *sprint* de Abril focou nos Locatários. Foram escolhidos os filtros, os formulários, os envios de e-mail e as permissões de usuário, permitindo assim que os locadores só consigam ver seus próprios apartamentos e quartos.

Sprint 5 (12 pontos) - Maio: Permitir o sign up de qualquer locador (1), Implementar permissão de administrador (1), Renderizar os alertas nas *views* corretas (1), Lei do Inquilinato disponível em uma aba do sistema (1), Desenvolver menu (1), Conexão com o S3 da Amazon para armazenar fotos de Quartos (3), Formulário de aviso de interesse quando não encontra um quarto que lhe interessa (1), Disponibilizar quartos livres mais recentes na página principal (3).

Motivação: Toques finais. O *sign u*p gratuito e simples para qualquer locador, a Lei do Inquilinato disponibilizada em uma aba do sistema, os alertas sendo renderizados de maneira correta em todo o sistema, o menu tanto para locatários quanto para locadores, a conexão com o S3 da Amazon para permitir *upload* de fotos e o último formulário do sistema.

4.4 Comparação entre Scrum points e Function points

Em projetos de desenvolvimento de software, análise de ponto de função é uma técnica que visa estabelecer uma medida em tamanho para que os gerentes tenham um conjunto de dados úteis e tangíveis para dimensionar, estimar, planejar e controlar projetos de software com rigor e precisão. Os pontos de função consideram as funcionalidades implementadas sob o ponto de vista do usuário. A medida é independente da linguagem de programação ou tecnologias que serão utilizadas para implementação.

A metodologia ágil de desenvolvimento SCRUM também possui maneiras de medir o esforço de desenvolvimento, entretanto, não há um gerente de projeto que será responsável por decidir como o software será medido e sim uma equipe consciente da importância da medição para que a mesma consiga auxiliar nos prazos e na execução de tarefas, seguindo o processo SCRUM.

Uma vez calculados os pontos de função de um sistema, cada ponto será equivalente a uma porção de uma funcionalidade que irá gerar um esforço em pessoas x horas e um custo para empresa. Uma equipe de desenvolvedores experientes atuando sobre esta funcionalidade do sistema não altera os pontos de função, o que não acontece na metodologia ágil SCRUM onde os *SCRUM points* são métricas relativas que mudam conforme os *sprints* realizados e com a expertise da equipe.

No desenvolvimento do projeto Imobiliária Web foi utilizada a metodologia SCRUM, que foi mais conveniente já que não existiu a contratação de desenvolvedores e por isso não foi necessário estabelecer uma medida de custo por função desenvolvida e sim uma medida de produtividade da equipe, para nós mesmos nos mantermos dentro do prazo que estipulamos.

5 IMPLEMENTAÇÃO USANDO RUBY ON RAILS

Este capítulo tem como principal objetivo demonstrar toda a parte técnica de implementação de um sistema utilizando o framework Ruby on Rails e servir de guia para aqueles que quiserem realizar a construção de um projeto utilizando esta linguagem.

5.1 Visão técnica geral

Para este projeto, foram adotadas as versões 2.1 do Ruby e 4.0.1 da *gem* Rails. *Gems* são bibliotecas ou programas em um formato padronizado que podem ser facilmente instaladas pelo gerenciador de pacotes RubyGems, para a linguagem de programação Ruby. O sistema foi desenvolvido a partir de uma distribuição Linux, o Ubuntu 14.04 LTS.

5.1.1 Ruby

Ruby é uma linguagem de programação dinâmica, reflexiva e orientada a objetos. Foi criada por Yukihiro “Matz” Matsumoto, no Japão [Matsumoto, 2002]. Ruby foi influenciada pelas linguagens Perl, Smalltalk, Eiffel, Ada e Lisp. Suporta múltiplos paradigmas da computação, incluindo os funcionais, orientados a objetos e imperativos, e tem um sistema de tipagem dinâmica e gerenciamento de memória automático.

5.1.2 Rails

Rails é um f*ramework model-view-controller* (MVC) que provê estruturas padrão para banco de dados, serviços web e páginas web. Ele encoraja e facilita o uso dos padrões da web, como XML ou jSON para transferência de dados, e HTML, CSS e JavaScript para exibição e interface de usuário. Além do MVC, Rails enfatiza o uso de alguns padrões de engenharia de software, como *Convention over Configuration* (CoC), *Don’t Repeat Yourself* (DRY) e o *Active Record*.

5.1.3 Model-View-Controller (MVC)

MVC, ou *model-view-controller*, é um padrão de arquitetura de software para implementação de interfaces de usuário. Ele divide uma aplicação de software em três partes (camadas) interconectadas, para diferenciar representações internas de

informação da maneira que o usuário as percebe. Estas camadas são: *View* (as telas, ou seja, o que de fato o usuário vê do sistema), *Controller* (os controladores da informação, ou seja, quem é responsável por fornecer as informações relevantes para cada *view*) e *Model* (onde fica toda a lógica do negócio da aplicação, e que é acessado pelos *controllers*).

5.2 Criação do Projeto e Implementação do Primeiro Sprint

5.2.1 Criação do Projeto

Para começar um novo projeto em Rails, dado que o Ruby e o Rails estão instalados no sistema, basta digitar no terminal o comando:

```
$ rails new nome-do-projeto
```

Uma vez feito isso, toda a estrutura padrão de uma aplicação Ruby on Rails será criada. O arquivo Gemfile lista todas as *gems* do projeto, inclusive a *gem* Rails, enquanto o arquivo Gemfile.lock armazena as versões de cada uma destas *gems*. A estrutura de pastas segue como descrito a seguir:

- App: onde são armazenados os *models, views, controllers, assets* (onde ficam armazenados os arquivos CSS e JavaScript), *helpers* (arquivos Ruby com métodos públicos para utilização em *views*) e *mailers* (arquivos responsáveis por envio de e-mail).
- Bin: onde são armazenados arquivos *default* de inicialização do Rails.
- Config: onde são armazenados todos os arquivos de configuração do projeto. Os arquivos *application* (configurações gerais da aplicação), *boot* (arquivo padrão de inicialização da aplicação), *environment* (arquivo padrão de ambientes da aplicação), *routes* (arquivo onde são armazenadas todas as rotas *restful* do projeto) e *database* (arquivo onde são armazenadas as configurações dos bancos de dados utilizados para cada ambiente da aplicação) estão na raiz da pasta. Além deles, existem três pastas: *environments* (onde estão armazenados os arquivos de configuração para cada ambiente utilizado pelo desenvolvedor, sendo estes geralmente *prodution*, *development* e *test*), *initializers* (onde estão armazenados

arquivos que devem ser carregados na inicialização da aplicação) e *locales* (onde estão armazenados os arquivos de internacionalização do Rails, ou seja, de línguas como Português, Inglês e etc.).

- Db: onde são armazenados todas as migrações do banco de dados do sistema, na pasta *migrations*, assim como um arquivo de *seeds* (com os dados necessários para o uso mínimo do sistema, prontos para serem inseridos quando a aplicação é lançada) e o *schema* (arquivo que contém toda a estrutura do banco de dados, com todas as tabelas, chaves estrangeiras e índices) em sua raiz.
- Lib: onde podem ser armazenados alguns dos *assets* do projeto, as tarefas *rake* e *templates* da tarefa rake já embutida no Rails, que é a *scaffolding* (não foi utilizada neste projeto, mas é a geração de models, views e controllers padronizados).
- Log: onde ficam armazenados os *logs* do sistema.
- Public: a pasta pública do projeto, que pode ser utilizada para armazenar todas as imagens do projeto (só foram colocadas aqui as imagens gerais, como para *footer* e *header*) e as páginas gerais de erro (404, 422 e 500).
- Test: a pasta padrão para os testes unitários do Rails.
- Vendor/assets: a pasta onde fica toda a pré-compilação dos *assets* do Rails.

O arquivo Gemfile contém todas as *gems* que serão usadas no sistema, e pode ser alterado durante o processo de desenvolvimento de acordo com as necessidades que os desenvolvedores encontram. Uma vez que o projeto acabou de ser criado, é necessário executar um comando que baixa e instala todas as *gems* listadas. Este é o mesmo comando que deve ser utilizado toda vez que uma nova *gem* é adicionada ao arquivo:

```
$ bundle install
```

5.2.2 Primeiro Sprint

No primeiro *sprint*, foram priorizadas as seguintes tarefas: configurar o banco de dados PostgreSQL, desenvolver a classe de usuários, desenvolver a *index* e fazer o *deploy* do sistema no Heroku [Henry, 2007].

Para configurar o banco de dados, basta entrar no arquivo `config/database.yml` e escrever o nome de usuário do banco, o nome dos bancos, a senha deste usuário e escolher o adaptar a ser utilizado (por padrão é o sqlite3), como segue:

```
development:
adapter: postgresql
encoding: unicode
database: meu-projeto-development
username: root
password: senha_root

test:
adapter: postgresql
encoding: unicode
database: meu-projeto-test
username: root
password: senha_root

production:
adapter: postgresql
encoding: unicode
database: meu-projeto-production
pool: 5
username: root
password: senha_root
```

Feito isto, é necessário executar o seguinte comando no terminal:

```
$ rake db:create
```

Ele criará todos os bancos especificados no arquivo, desde que os usuários e senhas estejam corretos e tenham permissão para mexer no PostgreSQL instalado na máquina ou servidor. Feito isso, criamos a classe User. Através do seguinte

comando, o Rails cria o modelo, a migração do banco de dados e os arquivos de teste responsáveis:

```
$ rails g model User email:string
```

Ou seja, estamos criando a classe User, sendo que o Rails entende que o nome das tabelas deve ser sempre no plural, criando assim uma migração para a tabela Users; o email:string representa o nome_do_campo:tipo_de_dado esperado. Poderíamos criar diversos campos nesta parte, todos separados por espaço, mas no momento foi decidido que criaríamos apenas isso, pois no futuro utilizaremos a *gem* Devise para tratar da autenticação de um usuário, e ela traz migrações pré-estabelecidas para a classe User.

A migração foi feita, mas precisa ser executada. Para tal, basta executar no terminal:

```
$ rake db:migrate
```

No próprio terminal será avisado se a migração foi rodada com sucesso ou se houve algum problema. É importante dizer que, uma vez que se queira preparar o banco em uma outra máquina, basta rodar rake db:setup que o banco será criado e todas as migrações executadas.

Para a criação da *index* do sistema, foi criado o arquivo `app/views/pages/home.html.erb`. A extensão `.erb` no fim da página html permite que se utilize código do Ruby on Rails em uma página HTML. Este arquivo não é o único responsável pela index, entretanto, pois existe o arquivo `app/views/layout/application.html.erb` que é executado em todas as páginas do sistema. Segue uma parte como exemplo:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8">
<meta http-equiv="X-UA-Compatible" content="IE=edge">
```

```
<meta name="viewport" content="width=device-width, initial-scale=1">
<title>Imobiliária Web</title>
<%= stylesheet_link_tag "application", media: "all" %>
<%= javascript_include_tag "application" %>
<%= csrf_meta_tags %>
</head>
<body>
<%= yield %>
</body>
</html>
```

A tag `<%= yield %>` é a responsável por trazer o conteúdo da página sendo de fato acessada. É importante ressaltar que poderíamos ter dado qualquer nome à pasta `app/views/pages`. Apesar de ela ter sido criada, ainda não existe uma rota para deixá-la acessível. Como toda rota aponta para uma ação dentro de um *controller*, precisaríamos criar um *controller* para exibir a nossa *homepage*. Entretanto, no início do projeto foi definida a adoção da *gem* HightVoltage uma biblioteca de geração de páginas estáticas [Thoughtbot, 2009], que irá gerar as páginas de conteúdo. Desta forma, adicionamos a *gem* ao arquivo Gemfile e executamos o comando mencionado anteriormente. Feito isto, seguindo a documentação da *gem*, criamos o arquivo `config/initializers/high_voltage.rb` e inserimos o código apontado.

Finalmente, podemos acessar nossa *homepage* através da url do projeto acrescido de `/home`. Este é o caso da nossa *homepage*, pois todas as outras páginas criadas dentro do diretório `/pages` precisará de `/pages/nome_do_arquivo`.

Para terminar as tarefas deste *sprint*, restou hospedar o sistema no Heroku [Henry, 2007]. Depois de ter feito uma conta gratuita e ter instalado o *git* no sistema, seguem os passos:

```
$ heroku login
$ heroku create
$ heroku push heroku master
$ heroku run rake db:setup
```

Com isso, o sistema está hospedado no Heroku e pronto para ser acessado. Lembrando que, toda vez que uma nova versão for enviada ao Heroku, será necessário rodar

```
$ heroku run rake db:migrate
```

Caso tenha alguma nova migração.

5.3 Segundo Sprint

No segundo sprint, foram priorizadas as tarefas de *layout* do sistema. Não será detalhado este processo, uma vez que foram utilizadas quase que unicamente as bibliotecas do Twitter Bootstrap [Twitter, 2010]; nada disso é específico do Rails, uma vez que esta estratégia pode ser aplicada sobre qualquer projeto *web*.

O que será detalhado é o *asset pipeline* da versão 4 do Rails. É um *framework* que concatena e minimiza os arquivos CSS e JavaScript, assim como permite a escrita destes arquivos em outras linguagens e pré-processadores, como CoffeeScript, Sass e ERB. Tecnicamente, o *asset pipeline* não faz mais parte do *core* do Rails, ou seja, não está dentro da *gem*, sendo uma *gem* por si só. Mas é uma dependência padrão do Rails, vindo assim habilitado por padrão. Para desabilitá-lo, basta adicionar a opção quando for criar a aplicação:

```
$ --skip-sprockets
```

A primeira característica do *asset pipeline* é concatenar os arquivos CSS e JavaScript, desta forma diminuindo a quantidade de chamadas que um *browser* faz ao servidor para renderizar uma página e, por consequência, deixando a aplicação mais rápida.

Todos arquivos CSS são reunidos em um único arquivo mestre .css, e a mesma coisa acontece para os arquivos JavaScript, reunidos em um arquivo .js. Em produção, o Rails por padrão aplica um cache sobre estes arquivos, sendo descartado automaticamente quando algum deles é modificado.

A segunda característica é minificar ou comprimir os arquivos. Esta minificação do arquivo CSS é feita removendo todos os espaços em branco e comentários; para o arquivo JavaScript, processos mais elaborados podem ser realizados. O desenvolvedor pode escolher qual processo prefere adotar ou criar o seu próprio em um arquivo de configurações, mas para este projeto foi utilizada a opção padrão.

5.4 Terceiro Sprint

Para o terceiro *sprint*, foram escolhidas as tarefas de CRUD. Depois que o primeiro CRUD é feito, os demais são praticamente iguais, facilitando assim o trabalho e permitindo a rápida execução das tarefas. Para não deixar esta seção repetitiva, será demonstrada apenas a criação completa do CRUD de Apartamentos, levando em consideração que a classe Quarto já está implementada, para que seja possível mostrar como os relacionamentos entre modelos funcionam no Rails.

Primeiro, fazemos como foi feito com a classe User:

```
$ rails g model Apartament neighborhood:string street:string number:string apartment:string dimensions:string bathrooms_quantity:integer living_rooms_quantity:integer service_areas_quantity:integer user:references
```

Nada novo por aqui, com exceção do último atributo. O parâmetro *references* serve para informar que o modelo gerado terá uma associação *belongs_to* com o atributo informado, ou seja, gerará um *user_id* e o modelo será criado com a seguinte linha de código já implementada:

```
$ belongs_to user
```

Isto significa que, em qualquer lugar do sistema, caso se tenha um objeto de algum apartamento instanciado, pode-se utilizar

```
$ objeto.user
```

e obter assim o usuário associado. Considerando o que temos em mente para o modelo de apartamentos, vejamos como ficou nosso modelo:

```
class Apartment < ActiveRecord::Base

      belongs_to :user
      has_many :rooms, :dependent => :restrict

      validates_presence_of      :neighborhood,      :street,      :number,
:bathrooms_quantity,    :living_rooms_quantity,    :service_areas_quantity,
:city_id

      def full_address
            if number.present? && apartment.present?
                  neighborhood + ", " + street + " " + number + ", Apt " +
apartment
            else
                  public_address
            end
      end

      def public_address
            neighborhood + ", " + street
      end

      def default_code
            number + apartment
      end
end
```

O *belongs_to* representa um relacionamento 1:1, enquanto o *has_many* um 1:N. Desta forma, levando em conta que temos um objeto de Apartment instanciado, conseguimos a coleção de quartos de um apartamento através de:

```
      $ objeto.rooms
```

Na classe Room, teremos um *belongs_to* também. Qualquer associação onde uma classe possui a chave estrangeira de outra, é entendido que esta pertence à outra, desta forma possuindo o *belongs_to*. No caso, *has_many* e *has_one* são usados pelos “donos” das outras classes, sendo que no primeiro se obtém os objetos associados através do nome da classe no plural e o segundo no singular.

Isto feito, temos nosso modelo pronto, com alguns métodos que julgamos interessantes para exibir os endereços completos e públicos, além de um código que adotamos como padrão para todo apartamento do sistema. Percebe-se que pode somar *strings* em Ruby, como fica evidenciado na implementação destes métodos.

Agora, para podermos criar, editar, remover ou visualizar um apartamento, precisaremos de um *controller* com todas estas ações implementadas. Para criá-lo, basta executar o seguinte comando:

```
$ rails g controller apartaments
```

Segue como ficou nosso *controller* de Apartaments, por ação com as explicações em seguida:

```
def index
      @apartments = current_user.apartments.order(:id)
end
```

A variável apartments está precedida por um @ para que ela possa ser acessada dentro da *View* correspondente desta ação. O método *current_user* será explicado posteriormente, quando a *gem devise* for explicada. Toda coleção e todo *array* tem o método *order* implementado, onde se passa como parâmetro o campo pelo qual a coleção será ordenada.

```
def show
      @apartment = Apartment.find(params[:id])
end
```

Por padrão, a variável *params* dentro de todo *controller* é um *hash* com os parâmetros passados para a ação onde se encontra. No caso, foi passado como parâmetro o *id* do apartamento em questão, possibilitando assim encontrá-lo da forma mais rápida possível. No Rails, todo *id* é indexado.

```
def new
      @apartment = Apartment.new
end
```

Na ação *new*, um novo apartamento é instanciado a partir do método *.new*. É possível criar construtores para as classes, que são chamados quando o *.new* é invocado, mas para a classe Apartment não foi necessário implementar um.

```
	def create
		@apartment = Apartment.new(apartment_params)
		@apartment.user_id = current_user.id
		if @apartment.save
			redirect_to apartments_url, :notice => "O apartamento foi
criado com sucesso."
		else
			render :new
		end
	end
```

Na ação *create*, há algumas coisas interessantes para mostrar. O método *new* é novamente chamado, desta vez com uma variável passada como parâmetro. Esta variável, como será evidenciado mais tarde, é um *hash* de valores. O método *new* aceita um *hash* para instanciar os objetos, lançando uma exceção caso exista alguma chave que não corresponda à nenhum atributo da classe. Temos um *if* no momento que o apartamento é salvo pois, caso apresente algum erro (como por exemplo o campo street vazio, uma vez que no apartamento está sendo validada a presença deste), é renderizada a ação *edit* sem fazer uma nova requisição ao servidor, desta forma mantendo os dados que foram preenchidos e exibindo as mensagens de erro.

Caso seja salvo, o *redirect_to* redireciona para a *index* de apartaments, com a mensagem contida no *notice*.

```
	def edit
		@apartment = Apartment.find(params[:id])
	end
```

Nada novo a acrescentar nesta ação.

```
	def update
		@apartment = Apartment.find(params[:id])
		if @apartment.update_attributes(apartment_params)
			redirect_to apartments_url, :notice => "O apartamento foi
atualizado com sucesso."
```

```
            else
                  render :edit
            end
      end
```

Muito semelhante ao método *create*, as únicas diferenças do *update* são o método *update_attributes* que faz exatamente o que o método sugere: atualiza os atributos de acordo com um *hash* de parâmetros; a outra é que é renderizada a ação *edit* e não a *new*.

```
      def destroy
            @apartment = Apartment.find(params[:id])
            if @apartment.destroy
                  redirect_to apartments_url, :notice => "O apartamento foi
removido com sucesso."
            else
                  redirect_to apartments_url, :alert => "O apartamento não
foi deletado; verifique se ele possui quartos."
            end
      end
```

Na ação *destroy*, achamos relevante apenas acrescentar que o método *destroy* chama os *callbacks* do Rails, enquanto o *delete* executa no banco de dados um comando DELETE, assim ignorando qualquer *callback* da aplicação.

```
      private
      def apartment_params
            params.require(:apartment).permit(:neighborhood,       :street,
:number,    :apartment,    :bathrooms_quantity,    :living_rooms_quantity,
:service_areas_quantity, :dimensions
end
```

No final de nosso *controller*, temos uma ação seguida de um *private*, ou seja, uma ação privada que só pode ser acessada dentro do próprio *controller*. Este método, o apartment_params, é um padrão do Rails 4 para evitar uma injeção de SQL na hora de criar ou editar um registro. Este código significa que, para um apartment, só são permitidos os parâmetros passados dentro do *permit*.

Agora que temos nosso modelo e nossas ações do *controller* implementadas, faltam apenas as telas e uma forma de avisar ao Rails que estas ações são acessíveis. As *views* ficarão em

`app>views>apartaments>nome_do_arquivo.html.erb`, ou seja, um arquivo HTML que interpreta código do Ruby on Ruby. As *views* precisam ter o mesmo nome das ações, ou seja, index.html.erb, news.html.erb e edit.html.erb. Para executar o código exibindo o resultado, basta colocar o código dentro de uma tag `<%= %>`; para executar mas sem exibir o resultado, `<% %>`.

Para habilitar as rotas, basta ir até o arquivo routes.rb e adicionar a seguinte linha de código:

```
$ resources :apartaments
```

Nós poderíamos dizer que só queremos que o *index* de apartments esteja disponível, que não é o nosso caso, mas para isso bastaria colocar:

```
$ resources :apartments, :only => [:index]
```

Da mesma forma, poderíamos dizer que queremos todas as ações, com exceção da *index*. Para isso:

```
$ resources :apartments, :except => [:index]
```

5.5 Quarto *Sprint*

Neste *sprint*, serão detalhadas as duas funcionalidades que exigiram uma maior atenção por parte dos desenvolvedores. A primeira delas é o envio de e-mail. Nós poderíamos gerar o *mailer* através de um comando, mas muitos arquivos que não serão utilizados seriam gerados por padrão, então foi decidido criar o arquivo manualmente em `app>mailers`. Como o envio de e-mail neste projeto está associado ao preenchimento de formulários, criamos o `form_mailer.rb`.

A seguir estarão partes do arquivo, explicando como os *mailers* funcionam no Rails.

```
default from: "usuario@dominio.com"
```

Esta linha significa que, por padrão, em todos os e-mails aparecerá que o remetente é o e-mail usuario@dominio.com.

```
def registry(formulario)
      @formulario = formulario
mail(to: "outro.usuario@dominio.com", subject: 'Registry Form')
end
```

Os *mailers* funcionam de maneira semelhante aos *controllers*, onde cada método é na realidade uma ação. A diferença é que estas não precisam ser declaradas no arquivo de rotas. Neste caso, a ação *registry* do *mailer* FormMailer enviará um e-mail ao some.email@gmail.com com o assunto Registry Form. O corpo do e-mail deverá estar em `app>views>form_mailer>registry.html.erb`.

Para configurar o e-mail do qual serão enviadas as mensagens, basta entrar no arquivo production.rb e adicionar o seguinte bloco de código:

```
config.action_mailer.smtp_settings = {
address: 'smtp.dominio.com',
port: 25,
domain: 'domínio',
user_name: 'usuario@dominio.com',
password: 'my_password',
authentication: 'plain',
enable_starttls_auto: true }
```

Desta forma, o envio de e-mails está configurado. Para enviar um e-mail através da ação *registry*, basta executar o seguinte comando:

```
FormMailer.registry(@formulario).deliver
```

Assim, dentro da *view*, o objeto @formulario estará acessível para ser utilizado para compor o corpo do e-mail.

A outra funcionalidade escolhida para ser evidenciada nesta seção é a do filtro de quartos por cidade, mês e ano. Segue a função implementada na classe Room:

```
def self.search(month, year, city_id)
      last_date_of_month = Date.new(year, month+1, 1) - 1.day unless month
== 12
      last_date_of_month ||= Date.new(year, 12, 31)
first_date_of_month = Date.new(year, month, 1)
      joins(:apartment).where("availability_date <= ? and availability_date
>= ? and apartments.city_id = ?", last_date_of_month, first_date_of_month,
city_id)
end
```

A primeira coisa importante de se mostrar é que o método começa com self., ou seja, é um método de classe e não de objeto. Desta forma, para executá-lo, será sempre necessário fazer Room.search(parametros). A segunda é o *unless*, que funciona justamente como um *if* invertido; é possível usar *if* e *unless* na própria linha. E por conta disso a segunda linha, ao invés de receber com = , recebe como ||=, ou seja, só recebe o valor a seguir caso a variável esteja vazia.

O joins(:apartment) faz um *join* com a classe Apartment através da chave estrangeira especificada na própria classe Room (belongs_to :apartment). Esse *joins* está solto pois, uma vez dentro de um método de classe, considera-se que um método destes está sendo executado dentro da classe; ou seja, self.where teria o mesmo resultado. Dentro do *where*, colocamos uma string SQL normal, cuja única parte diferente são os pontos de interrogação, que ficam reservados para serem substituídos pelas variáveis a seguir da *string*.

5.6 Quinto *Sprint*

Para o último *sprint* do projeto, serão detalhadas três funcionalidades: a conexão com o S3 da Amazon para hospedar as imagens, a implementação do *Devise* que trata de autenticação e *sign-up* de usuários e, por último, a implementação do CanCan, que trata das permissões de usuário.

Diversas *gems* podem auxiliar na conexão com a S3 da Amazon, mas para o projeto foi adotada a *gem carrierwave*. Após adicioná-la no *Gemfile* e executar o comando para instalá-la, devemos executar o seguinte comando:

```
$ rails g uploader Photo
```

Assim, um arquivo é gerado em `app>uploaders`. No nosso caso, foi o photo_uploader.rb. O arquivo já vem com muita coisa preenchida, e basta comentar a linha de código que aponta que as fotos serão armazenadas em arquivo e descomentar a que afirma que eles serão armazenadas em fog.

Feito isto, basta ir em config>initializers e criar o arquivo `carrierwave.rb`. Nele, colocar as credenciais da Amazon, da seguinte forma:

```
CarrierWave.configure do |config|
config.fog_credentials = {
:provider => 'AWS',
:aws_access_key_id => ‘AWS_ACCESS_KEY_ID’,
:aws_secret_access_key => 'AWS_SECRET_ACCESS_KEY'
}
config.fog_directory = 'my_bucket'
end
```

Assim, quase todo trabalho está feito. Agora basta rodar uma migração para o modelo que terá uma foto, desta maneira:

```
rails g migration add_avatar_to_rooms photo:string
```

Assim o atributo photo guardará todas informações necessárias para ligar o arquivo que estará hospedado no S3 da Amazon com a informação contida no banco de dados. Feito isto, adicionar a seguinte linha no modelo Room:

```
mount_uploader :photo, PhotoUploader
```

E toda a infraestrutura para armazenar arquivos no S3 da Amazon para os quartos do sistema está pronta.

A gem CanCan é uma biblioteca de autorização para Ruby on Rails, que restringe os recursos e permissões que um determinado usuário possui [Ryan, 2010] é extremamente útil e fácil de ser manipulada, reunindo todas as permissões de usuário em um único arquivo. Para gerá-lo, basta executar o seguinte comando:

```
$ rails g cancan:ability
```

É possível criar diversos perfis de usuário, mas como nosso projeto não necessita de tanto, abaixo está demonstrado como ficou o arquivo gerado em `app>models>ability.rb`:

```
class Ability
include CanCan::Ability

def initialize(user)
user ||= User.new # guest user (not logged in)
if user.admin?
can :manage, :all
end
end
end
```

Isto significa que, se o usuário é *admin*, ele tem permissão completa sobre os seus apartamentos e quartos; nenhum usuário poderá visualizar, editar ou excluir apartamentos e quartos que não sejam seus, ou seja, que não estejam com seu *user_id.*

Além disso, para esconder um botão ou parte do sistema de acordo com a permissão, basta adicionar o seguinte bloco de código dentro de uma *view*:

```
<% if can? :update, @aparment %>
<%= link_to "Edit", edit_apartment_path(@apartment) %>
<% end %>
```

Dependendo da ação, basta trocar o *:update* por *:create, :read ou :delete*. Nada disto funcionará a não ser que seja adicionado, dentro de cada *controller*, a seguinte linha de código:

```
load_and_authorize_resource
```

Desta forma, o CanCan estará habilitado para cada *controller* que possuir este código. Como nem todas as páginas do sistema precisam necessitam de autorização, este código foi adicionado apenas nos lugares relevantes.

A *gem* Devise é uma solução de autenticação flexível para Rails [Neighman, 2009] é extremamente versátil, mas apenas uma pequena parcela de seu potencial foi utilizado no projeto, dado sua baixa complexidade. Depois de adicionada ao *Gemfile* e instalada, o primeiro passo é executar o seguinte comando:

```
$ rails generate devise:install
```

Seria possível gerar o modelo *User* com todas as funcionalidades do *Devise*, mas como ele já foi criado no primeiro *Sprint*, será preciso rodar uma migração específica para adicionar os campos necessários. Seja qual for o nome da migração, ela precisará ter o seguinte conteúdo:

```
class NomeDaMinhaMigracao < ActiveRecord::Migration
      def change
            change_table :users do |t|
                  t.encrypted_password
                  t.confirmable
                  t.recoverable
                  t.rememberable
                  t.trackable
                  t.token_authenticatable
                  t.timestamps
            end
      end
end
```

Dentro da classe *User*, adicionar a seguinte linha:

```
devise :database_authenticatable, :recoverable, :rememberable, :trackable, :validatable, :registerable
```

Estes são módulos que o *Devise* oferece, mas não são todos; para melhor entendimento, visitar a documentação oficial da *gem* no Github. Agora temos implementado pelo *devise* o método *current_user*, que foi exibido anteriormente. Ele verifica se existe algum usuário logado no atual *browser* e, caso exista, retorna o próprio com este método.

Para fazer uso das *views* padronizadas do Devise, basta executar o seguinte comando no terminal:

```
$ rails generate devise:views
```

De acordo com a complexidade e o tamanho desta biblioteca, um capítulo inteiro poderia ser dedicado a ela, mas, como foi dito anteriormente, nós utilizamos uma pequena parcela de seu potencial. Para resumir do que ela é capaz, e como ela agiliza o desenvolvimento de um projeto, será exibido como um usuário se cadastra no sistema e como as rotas do *devise* são permitidas.

No arquivo de rotas, basta acrescentar:

```
devise_for :users
```

Assim, todas as rotas de todos os módulos explicitados na classe de usuários fica habilitada. Segue o código que está no `application.html.erb`, mostrando como é fácil tratar de *sign-up, sign-in* e *sign-out* de usuários:

```
<% if current_user.present? %>
<%= link_to "Mude sua senha", edit_user_registration_path %></li>
<%= link_to "Sair", destroy_user_session_path %></li>
<% else %>
<%= link_to "Registre-se", new_user_registration_path %></li>
<%= link_to "Entrar", new_user_session_path %></li>
<% end %>
```

Todas estas rotas e telas já vêm feitas pelo *devise*, podendo ser customizadas em `app>views>devise` da forma que a equipe achar melhor.

A seguir, é mostrado o resultado final do desenvolvimento com as principais telas do sistema Imobiliária Web.

Figura 7 - Tela Principal

A imagem abaixo mostra o formulário de cadastro de quartos que deve ser cadastrado pelo usuário referente ao seu imóvel.

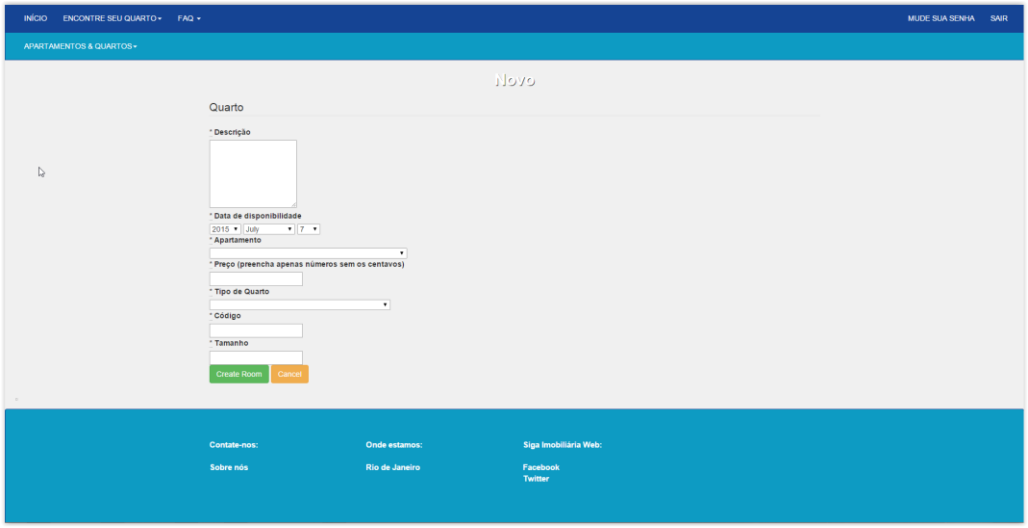

Figura 8 - Cadastro de Quartos

Na próxima tela, o usuário do sistema Imobiliária Web poderá efetuar o cadastro de reserva de quartos caso ele deseje ser notificado quando uma vaga estiver disponível.

Figura 9 - Formulário de Reserva

Estas são as funcionalidades mais complexas existentes no sistema Imobiliária Web, desenvolvidas ao longo dos cinco meses e divididos em cinco sprints.

O uso do framework Ruby on Rails apresenta inúmeras vantagens que foram observadas ao longo do desenvolvimento deste projeto sendo elas:

- Abstração de conhecimentos de query SQL para construção do banco de dados devido ao Active Record que é uma camada de mapeamento objeto-relacional.
- É uma linguagem de rápido desenvolvimento, já que o framework utiliza dois conceitos que visam aumentar a produtividade: o *DRY* e *Convention over Configuration*, conceitos que buscam a redução de repetição de código e tempo de desenvolvimento, fazendo com que o desenvolvedor dê preferência a utilização de componentes já existentes, amplamente usados e testados pela comunidade.
- É uma linguagem de código aberto possibilitando que um desenvolvedor seja capaz de alterar funcionalidades de acordo com a sua necessidade.
- Possui forte adesão da comunidade com usuários ativos e participativos.

Também foram observados pontos negativos sobre o framework elas são descritas abaixo:

- Configuração de um ambiente de desenvolvimento complicado já que existem inúmeras versões do framework e caso um projeto seja migrado de uma versão para outra erros poderão ocorrer no projeto.
- Difícil de se programar no MS Windows pois a instalação do framework nesta plataforma é complexa e com inúmeros *bugs* relatados.
- Plataformas de hospedagem de projetos em Ruby on Rails são mais caras que as plataformas usadas para outras linguagens.

Fica claro que o desenvolvimento de pequenas aplicações em Ruby on Rails, com o auxílio das *gems* certas, é simples e rápido, abstraindo toda complexidade de montar um banco de dados e pensar em uma modelagem, graças às migrações, que foram feitas diversas vezes ao longo do projeto.

6 CONCLUSÃO

Neste projeto, é possível observar todos os passos necessários para a construção de um sistema de informação que utiliza técnicas mais recentes do mercado de trabalho, tais como o uso de metodologias ágeis e linguagem de programação utilizando o *framework* Ruby on Rails. No desenvolvimento do projeto, foi possível observar inúmeras vantagens para a implementação em curtos períodos de tempo e poucos recursos humanos.

O desenvolvimento do sistema envolveu desde o levantamento de requisitos e modelagem de casos de uso e de classes até a implementação usando o framework Ruby on Rails. A conjugação do uso deste framework com as práticas do Scrum resultou em um desenvolvimento rápido e eficaz do sistema, que pode ser replicado em outras aplicações.

Este trabalho poderá servir de guia para aqueles que desejam conhecer uma pouco do framework Ruby on Rails e aprender o uso das principais g*ems*, o uso do sistema Heroku para *deploy* de sistemas de maneira prática, a customização do *front-end* com auxílio do Twitter Bootstrap e o uso do Devise, uma ferramenta versátil para autenticação.

No futuro, o projeto Imobiliária Web irá buscar uma popularização em seu uso no intuito de obter investimentos em patrocínios ou de venda de informações para grandes empresas imobiliárias que buscam aprimorar sua base de inteligência sobre o perfil de usuários que estejam apenas querendo alugar um quarto em imóvel.

O sistema Imobiliária Web estará no ar até o final de 2015, aproveitando a realização das Olimpíadas do Rio de Janeiro. Uma possível continuação deste projeto seria a implementação de todos os testes unitários e os testes de integração, que não foram previstos no escopo do sistema.

BIBLIOGRAFIA

Beck, Kent (2001) "Manifesto Ágil", http://www.manifestoagil.com.br/, Janeiro 2015.

Globo (2015) "FIPE-ZAP:ÍNDICE" http://www.zap.com.br/imoveis/fipe-zap-b, Maio 2015.

Hansson, H. David (2003),"Ruby on Rails", http://rubyonrails.org, Fevereiro 2015.

Henry, Orion. (2007),"Heroko", https://www.heroku.com , Junho 2015.

Matsumoto, Yukihiro . (2002). Ruby In A Nutshell, Edition: 1st.

Neighman, Daniel (2009) "Gem Divise", https://github.com/plataformatec/devise, Junho 2015.

Schwaber, K. & Sutherland, J. (2013). "Guia do Scrum - Um guia definitivo para o Scrum: As regras do jogo". http://www.scrumguides.org/docs/scrumguide/v1/scrum-guide-us.pdf, Junho 2015.

Thoughtbot. (2009) "HighVoltage", https://github.com/thoughtbot/high_voltage, Janeiro 2015.

Twitter. (2010), "Bootstrap", http://getbootstrap.com, Maio 2015.

Smaldone, Javier (2008) "RailRoady", https://github.com/preston/railroady, junho 2015.