UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

JEAN CARLO EMER

# Migrando Aplicações WEB para Plataformas Abertas: um Estudo de Caso

Trabalho de Graduação.

Prof. Dr. Carlos Alberto Heuser
Orientador

Porto Alegre, maio de 2013.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Carlos Alexandre Netto
Vice-Reitor: Prof. Rui Vicente Oppermann
Pró-Reitora de Graduação: Profa. Valquiria Link Bassani
Diretor do Instituto de Informática: Prof. Flávio Rech Wagner
Coordenador do CIC: Prof. João César Netto
Bibliotecária-Chefe do Instituto de Informática: Beatriz Regina Bastos Haro

# SUMÁRIO

# LISTA DE ABREVIATURAS E SIGLAS

| | |
|---|---|
| API | Application Programming Interface |
| FTP | File Transfer Protocol |
| HMVC | Hierarchical Model-view-controller |
| JSON | JavaScript Object Notation |
| MVC | Model-View-Controller |
| ODBC | Open Database Connectivity |
| ORM | Object Relational Mapper |
| PHP | Hypertext Processor |
| SGBD | Sistema de Gerenciamento de Banco de Dados |
| SQL | Structured Query Language |
| SSH | Secure Shell |
| UFRGS | Universidade Federal do Rio Grande do Sul |
| URL | Uniform Resource Locator |

# LISTA DE FIGURAS

# LISTA DE TABELAS

# RESUMO

O trabalho trata-se de um estudo de caso de migração de uma aplicação ASP.NET e Microsoft SQL Server para uma arquitetura de código aberto. A aplicação em questão tem como objetivo publicar informações históricas coloniais do Estado do Rio Grande do Sul mesclando um serviço WEB de mapeamento geográfico com técnicas de busca de informações por similaridade.

A aplicação, além de desenvolvida com tecnologias proprietárias, carecia de uma organização de código e estava calcada na API do Google Maps JavaScript v2, já descontinuada no momento da escrita deste trabalho.

O processo de migração atende aos mesmos requisitos da aplicação base com aprimoramentos de interface e funcionalidade. O produto final é consolidado em uma total reescrita da aplicação se utilizando de arquiteturas Open Source, Design Patterns e Code Standards.

**Palavras-Chave:** migração, Open Source, mapeamento, Design Patterns, Code Standards.

**Migrating Web Applications to Open Platforms: a Case Study**

# ABSTRACT

The current paper is a study of a migration from an ASP.NET and Microsoft SQL Server to an open source architecture. The application objective is to publish colonial historical information about Rio Grande do Sul with a web map service with an information search for similarities.

The application, built with proprietary tecnologies, needed a code organization and was sustained on Google Maps API JavaScript v2, already discontinued at the time of this working writing.

The migration proccess answers to the same base application requirements with improvements of interface and funcionality. The final product is consolidated in a full application rewriting using Open Source architectures, Design Patterns and Code Standards.

**Keywords:** migration, Open Source, mapping, Design Patterns, Code Standards.

# 1 INTRODUÇÃO

O uso de aplicações WEB têm se tornado cada vez mais recorrente com a popularização da internet. Segundo o Ibope Media, somos 94,2 milhões de internautas tupiniquins (dezembro de 2012). Juntamente com este crescimento, novas técnicas e ferramentas vão se aperfeiçoando e ganhando espaço.

O trabalho trata-se de um estudo de caso de migração de uma aplicação proposta e desenvolvido por (DOS SANTOS, 2009), denominada neste texto como Geo Colony 1.0, com objetivo de publicar informações históricas coloniais do Estado do Rio Grande do Sul mesclando um serviço WEB de mapeamento geográfico com técnicas de busca de informações por similaridade. A aplicação Geo Colony 1.0 em questão é escrita em ASP.NET (SCOTT, 2013) utilizando bando de dados Microsoft SQL Server (SQL SERVER. 2013), ambas tecnologias proprietárias da empresa Microsoft. Além disto, para o referenciamento geográfico, a API do Google Maps JavaScript v2, atualmente descontinuada, foi utilizada.

## 1.1 Objetivo

O objetivo deste trabalho é analisar e justificar o processo de migração da aplicação para tecnologias de código aberto, atendendo aos mesmos requisitos da aplicação base com aprimoramentos e novas funcionalidades.

Ao longo do texto o trabalho defende que, das muitas aplicações que executam em um servidor, as que demandam menos custos para serem mantidas são as desenvolvidas com tecnologia de código aberto e justifica a necessidade de migração neste caso específico.

O produto final é consolidado em uma total reescrita da aplicação utilizando no *back end* a linguagem de programação PHP (PHP, 2013) e o SGBD MySQL (MYSQL, 2013). O *front end* da aplicação utiliza o pré-processador de CSS denominado Stylus (KHAN, 2013) e a linguagem CoffeeScript (COFFEESCRIPT, 2013) que compila para JavaScript (MOZILLA, 2013). A API de mapas é a mais nova versão 3.0 da API do Google Maps JavaScript.

Todo o código desenvolvimento utiliza *Design Patterns* e *Code Standards*. Muito além da simples análise de uma migração, a intenção do trabalho é servir como guia de boas práticas incluindo uma vasta gama de ferramentas e tecnologias para garantir qualidade do produto final de *software*.

## 1.2 Organização do texto

A organização deste trabalho estabelece seis diferentes capítulos para descrever o problema, objetivo, ferramentas e técnicas utilizadas. Após a introdução, o capítulo dois

apresenta os conceitos, técnicas e serviços relacionados. Além disto, o capítulo apresenta a aplicação base, proposta por (DOS SANTOS, 2009), desta migração ilustrando suas características e funcionalidades.

A modelagem e estrutura da aplicação são apresentados no capítulo três, a identificação do problema e as funcionalidades da aplicação base iniciam este capítulo destinado a apresentar as Histórias de Usuário e arquitetura geral da aplicação.

O capítulo quatro foca na implementação da aplicação destacando os diferentes diagramas de classes e modelagens assim como o uso das tecnologias e resultados alcançados com cada uma delas.

A migração dos dados da aplicação base é apresentado no capítulo cinco, esquematizando o processo de uso das ferramentas aplicadas.

Por fim, no capítulo seis, a conclusão do trabalho é apresentada.

# 2 INSUMOS E CONCEITOS RELACIONADOS

Este capítulo tem a função de ilustrar alguns dos conceitos, técnicas, ferramentas, e serviços relacionados. A título de organização, as ferramentas serão divididas entre *back end* e *front end*.

O *back-end*, conhecido como lado servidor, é a porção do programa executado na máquina servidora da aplicação. A única maneira de ver o código *back end* é tendo acesso via SSH ou FTP ao computador. Em aplicações WEB a computação é disparada por uma requisição HTTP do cliente e seu resultado é o retorno a esta chamada.

O *front end*, também chamado de lado do cliente, é aquele mostrado para o usuário permitindo que o mesmo interaja diretamente e, em aplicações WEB, acessível através do código fonte da página.

É importante destacar que algumas tecnologias podem ser usadas no *front end* e *back end*. Este é o caso, por exemplo, do JavaScript que além de presente nos *browsers* está no servidor através do Node.js (NODEJS). Neste texto, a título de organização, a divisão foi baseada restritamente ao uso da tecnologia na aplicação.

Além disto, o capítulo apresenta a aplicação Geo Colony v 1.0, aplicação base desta migração, proposta por (DOS SANTOS, 2009), mostrando suas características e funcionalidades.

## 2.1 Code Standards

*Code Standards* é um guia para uma linguagem de programação com a função de indicar aos desenvolvedores como eles devem escrever código. Estas convenções podem indicar organização de arquivos, endentação, comentários, espaçamentos e quebras de linha entre outros aspectos.

Adotar *Code Standards* garante que grandes aplicações sejam programadas com um estilo consistente, o que, além de tornar o código mais fácil de se entender, permite que desenvolvedores que observem uma porção do código possam conhecer tudo aquilo que irão encontrar no restante da aplicação.

## 2.2 Design Patterns

Na Engenharia de *Software*, *Design Pattern* é uma solução geral e reutilizável para um problema recorrente em um dado contexto do *Design* de *Software*.

### 2.2.1 Model-view-controller (MVC)

O Model-view-controller é um padrão orientado a objetos de arquitetura de software que separa a matéria da aplicação da interface com o usuário. Este conceito foi introduzido por Trygve Reenskaug no Smalltalk-76.

A implementação clássica é dividida em três tipos de componentes e interações entre os mesmos, conforme descrito em (BUSHMANN, 1996).

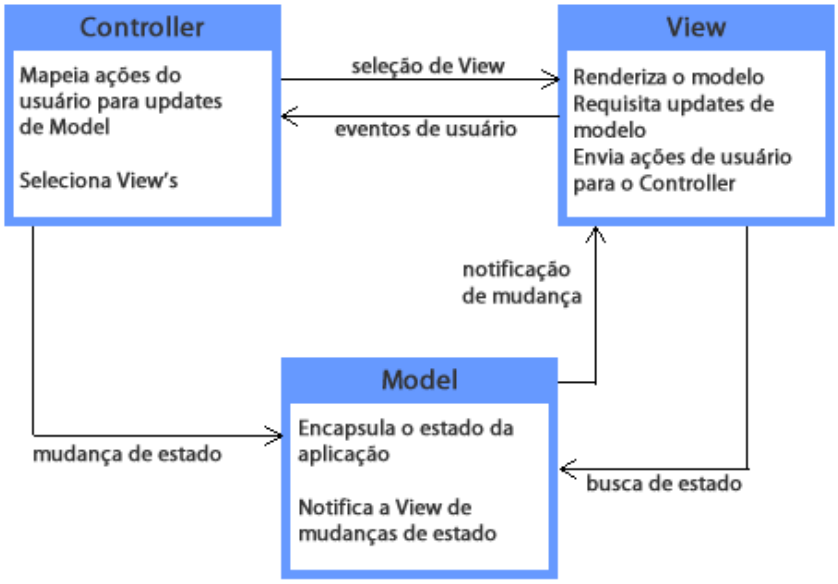


Figura 2.1: Diagrama de colaboração de um modelo MVC.

Fonte: Sommerville (2009, p.156)

**Controller** - pode enviar comandos para suas *views* associadas para mudar a representação de um *model*. Outra função do *Controller* é o envio de comandos para o *model* para que este possa mudar o seu estado.

**Model** - representa o conhecimento do domínio da aplicação respondendo a requisições de informações a respeito do seu estado e por instruções para alterá-lo. Uma implementação não passiva de *Model* é responsável também por notificar as *views* e *controllers* associados quando há mudanças em seu estado.

**View** - requisita para o modelo a informação que necessita para gerar um *output* agindo como um filtro de representação.

*2.2.1.1 Hierarquia de dependência*

A hierarquia descrita a seguir tem a função de minimizar as dependências entre os diferentes objetos do projeto.

A entidade mais isolada do padrão é o *Model* que unicamente conhece a si mesmo e seu código não deve fazer referência a *View* ou *Controller*. A *View* sabe apenas a respeito do *Model*, e requisita ao mesmo qual seu estado para saber o que precisa ser renderizado. Por fim, o *Controller* conhece a estrutura completa da aplicação.

*2.2.1.2 Aplicações WEB*

O modelo MVC foi extensivamente adaptado para aplicações WEB. As tecnologias utilizados nesta migração possuem diferentes visões sob os conceitos da aplicação clássica que serão ilustradas no capítulo 4.

### 2.2.2 Hierachical Model-view-controller (HMVC)

Variação do padrão MVC que permite a criação de aplicações mais sofisticadas através de um mecanismo que permite certas funcionalidades serem encapsuladas em seu próprio ecossistema de *Model*, *View* e *Controller*.

O principal benefício desta arquitetura é o ganho em modularidade, organização e principalmente reutilização de código.

## 2.3 Representational State Transfer (REST)

Trata-se de um estilo de arquitetura para aplicações que tira proveito das tecnologias e protocolos da WEB. O estilo define como os dados podem ser definidos e endereçados.

## 2.4 Google Maps

O Google Maps (GOOGLE, 2013) é um serviço de visualização de mapas e imagens de satélite gratuito na WEB fornecido pela empresa estadunidense Google.

O serviço oferece uma API que permite a criação de aplicativos baseados em localização, criação de mapas para aplicativos móveis, visualização de dados geoespaciais dentre outros. A distribuição é gratuita para uso não-comercial e pode ser incorporada a qualquer página WEB.

A API permite a criação de objetos polígonos que consistem em uma série de coordenadas em uma sequência ordenada. Tais formas podem ser utilizadas para demarcar limites de territórios.

## 2.5 GitHub

O GitHub é um serviço de hospedagem de *softwares* em desenvolvimento que fazem uso do controlador de versões distribuído GIT (GIT, 2013). Famoso por hospedar os mais populares projetos de código aberto, o serviço é usado para atrair comunidades de desenvolvedores e possibilitar que estes possam colaborar com os mais variados produtos.

## 2.6 Busca de dados por similaridade

Dados históricos provindos de várias fontes são carregados de erros tipográficos em nomes de pessoas e lugares. Pesquisas nestes dados com comparação exata podem levar a um resultado não tão relevante quanto o esperado. A solução adotada na aplicação, explicada com riqueza de detalhes em (DOS SANTOS, 2009), é a busca por aproximação de *strings* proposta em (GRAVANO, 2005).

### 2.6.1 Solução simplista

Bancos de dados relacionais não possuem suporte embutido para busca por aproximação de *strings*.

Uma abordagem é o emprego de UDFs (*User Defined Functions)* para calcular esta aproximação. A função *edit_distance(s1, s2, k)* necessita então ser registrada na base, seu resultado deve ser verdadeiro se as duas *strings* utilizadas como argumento estiverem dentro de uma distância de edição do argumento de número inteiro *k.* A seguinte SQL ilustra a solução:

```
SELECT      R1.Ai, R2.Aj
FROM        R1, R2
WHERE       edit_distance(R1.Ai, R2.Aj, k)
```

Contudo, tal abordagem é bastante ineficiente pois ao avaliar a expressão, que inclui múltiplas tabelas, o produto cartesiano das tabelas *R1* e *R2* deve ser resgatado e processado pela UDF. Por estas razões, a técnica implementada busca uma melhor solução, apresentada a seguir.

### 2.6.2 Preparação do banco de dados

Segundo (DOS SANTOS, 2009), para permitir o processamento aproximado de *strings* em um banco de dados através do uso de *q-grams*, é necessário um mecanismo para popular a base com os registros *q-grams* posicionais correspondentes às *strings* originais do banco de dados.

O processo inclui a criação de tabelas com o sufixo *_qgram* com as colunas *id*, *position* e *qgram*. De maneira prática, para o caso de um registro cuja coluna que se deseja pesquisar contenha o valor "geo_colony", seus *q-grams (position, qgram)* posicionais de tamanho q=3 são: (1, ##g), (2, #ge), (3, geo), (4, eo_), (5, o_c), (6, _co), (7, col), (8, olo), (9, lon), (10, ony), (11, ny$), (12, y$$). Então, a tabela com sufixo *_qgram* irá armazenar doze registros contendo em cada um deles o *id* do registro original seguido da *position* e *qgram* mostrados anteriormente.

### 2.6.3 Expressão SQL utilizando a técnica de Qgrams

O objetivo a ser alcançado é possível apenas com o uso de três diferentes técnicas de filtragem baseadas nas propriedades dos *q-grams*: filtro por contagem, filtro por posição e filtro por tamanho. Tais filtros podem ser expressos naturalmente como uma expressão SQL na base de dados conforme mostrado na Figura 2.2.

```
SELECT    R1.A0 , R2.A0 , R1.Ai , R2.Aj
FROM      R1 , R1AiQ , R2 , R2AjQ
WHERE     R1.A0 = R1AiQ.A0 AND
          R2.A0 = R2AjQ.A0 AND
          R1AiQ.Qgram = R2AjQ.Qgram AND
          |R1AiQ.Pos − R2AjQ.Pos| ≤ k AND
          |strlen(R1.Ai) − strlen(R2.Aj)| ≤ k
GROUP BY  R1.A0, R2.A0, R1.Ai, R2.Aj
HAVING    COUNT(*) ≥ strlen(R1.Ai) − 1 − (k − 1) ∗ q AND
          COUNT(*) ≥ strlen(R2.Aj) − 1 − (k − 1) ∗ q AND
          edit_distance(R1.Ai, R2.Aj, k)
```

Figura 2.2: Expressão SQL contendo as três propriedades dos *q-grams* (GRAVANO, 2005)

Algumas notações são necessárias para o decorrer da explicação. Usa-se *R* para denotar tabelas, *A* para denotar atributos de uma tabela. A notação $R_x.A_y$ referencia o atributo $A_y$ da tabela $R_x$. Usa-se *Q* para denotar a tabela auxiliar com sufixo *_qgram* correspondente. O *k* constitui o valor pelo qual a distância entre as *strings* que se deseja buscar deve ser inferior ou igual.

A expressão da Figura 2.2, realiza a junção das tabelas que se deseja comparar com suas respectivas tabelas auxiliares que contém os valores de *q-gram*. Em seguida, os *qgrams* das tabelas auxiliares estabelecem igualmente uma junção.

O filtro por posição é implementado como uma condição da cláusula WHERE eliminando quaisquer pares de *qgrams* em comum que estabeleçam uma distância superior a desejada.

O filtro por tamanho é implementado através da próxima condição da cláusula WHERE que simplesmente compara o tamanho das *strings*.

Estes filtros juntos reduzem o tamanho da junção dos *q-grams*, deixando a computação da expressão mais rápida, uma vez que menos pares de *q-grams* devem ser examinados pelas cláusulas seguintes.

O filtro de contagem é implementado pelas condições na cláusula HAVING. Os pares de *strings* que compartilham somente alguns *q-grams* são eliminados pelas expressões COUNT(*).

Por fim, mesmo após os passos de filtragem, o conjunto candidato pode ainda conter falsos positivos. Segundo (DOS SANTOS, 2009), deste modo a custosa invocação UDF *edit_distance*($R_1.A_i$, $R_2.A_j$, *k)* ainda deve ser realizada, mas, desta vez, em apenas uma pequena fração de todos os pares possíveis de *strings*.

## 2.7 Geo Colony 1.0

O sistema de busca e exibição de dados georreferenciados, denominado neste trabalho como Geo Colony 1.0, é fruto do trabalho de conclusão (DOS SANTOS, 2009). A aplicação faz uso do Google Maps como ferramenta de visualização das regiões coloniais do século XIX em um mapa. O resultado final oferece ao usuário uma interação intuitiva com os lotes coloniais e sua representação geográfica.

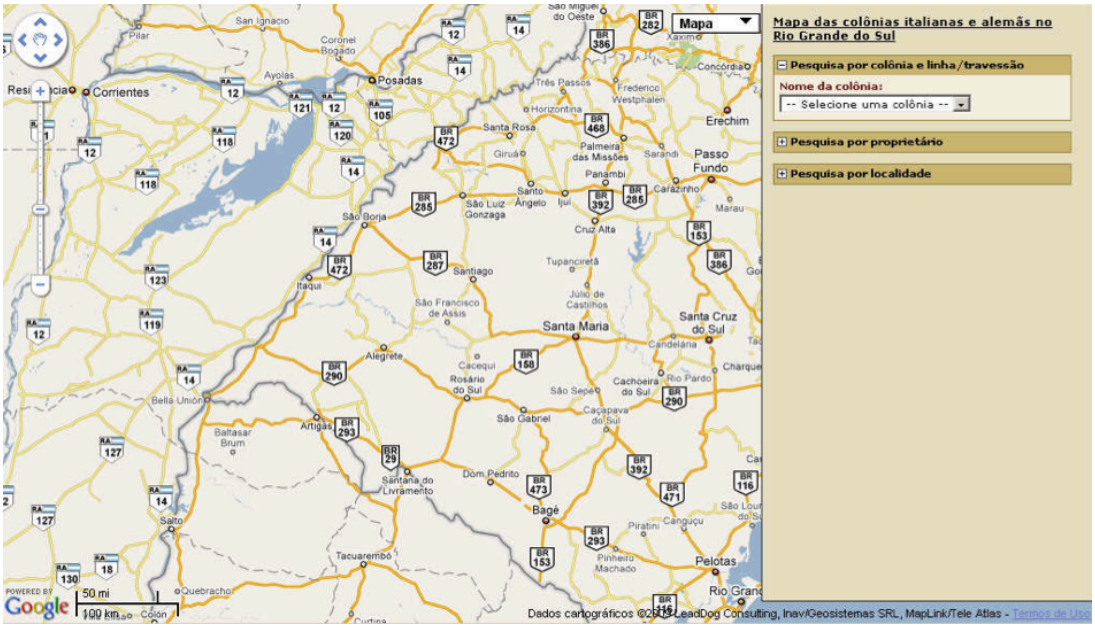


Figura 2.3: Tela da interface inicial da aplicação Geo Colony 1.0.

Fonte: Dos Santos (2009, p.25)

### 2.7.1 Terminologias

A aplicação introduz uma série de termos típicos do domínio de negócio que são essenciais para a compreensão do texto que segue. As definições a seguir foram extraídas do website da Prefeitura de Caxias do Sul - RS.

**Colônias:** Locais onde eram estabelecidos os territórios de imigrantes. Toda a colônia possuía uma Diretoria da Colônia que constituía a sede administrativa da colônia.

**Linhas ou travessões:** Caminhos traçados no meio da mata com seis a treze quilômetros de extensão que serviam como divisores dos lotes. Os travessões agrupados formavam as léguas.

**Lotes**: Surgidos dentro das linhas e travessões, a legislação estabelecia que suas extensões fossem entre 22 e 25 hectares. Na prática, seus tamanhos variavam em decorrência da topografia tendo alguns com até 80 hectares.

### 2.7.2 Tecnologias utilizadas

A aplicação Geo Colony 1.0 foi construída utilizando HTML, CSS, JavaScript XMLHttpRequest e JSON. Todas estas, tecnologias utilizadas na nova versão da aplicação e portanto descritas a seguir neste mesmo capítulo.

As seguintes tecnologias se diferem e portanto as analisaremos para facilitar o entendimento e antever argumentos para justificar diferentes escolhas tomadas no desenvolvimento da nova aplicação.

**ASP.NET** - A implementação é escrita em linguagem C#, uma daquelas suportadas pelo framework WEB *server-side* ASP.NET desenvolvido pela Microsoft sob uma licença proprietária. O framework possui como dependência o *Internet Information Server* que é licenciado apenas para máquinas Windows.

**ASP.NET AJAX** - Conjunto de extensões do *framework* ASP.NET com a finalidade de fornecer uma interface de programação Ajax simplificada e produtiva.

**Microsoft SQL Sever** - Sistema de Gerenciamento de Base de Dados relacional desenvolvido pela Microsoft. A solução proposta também oferece suporte a MySQL, porém a instância em produção não utiliza este sistema.

### 2.7.3 Modelo de desenvolvimento

O modelo adotado é centrado no usuário, tal se caracteriza pelo fato da lógica de apresentação e de negócio da aplicação ser gerenciada através de linguagem JavaScript no *browser* do usuário. As interações entre cliente e servidor limitam-se apenas ao acesso dos dados estritamente necessários.

### 2.7.4 Modelo de dados

Segundo (DOS SANTOS, 2009), todas as informações foram cedidas por Otavio Augusto Boni Licht, geólogo, mestre e doutor, da L&S Consultoria Geológica Ltda, de Curitiba, PR. Os dados foram georreferenciados através do software ArcGIS e, para serem utilizados pela aplicação, foram convertidos de forma que pudessem popular um banco de dados relacional.

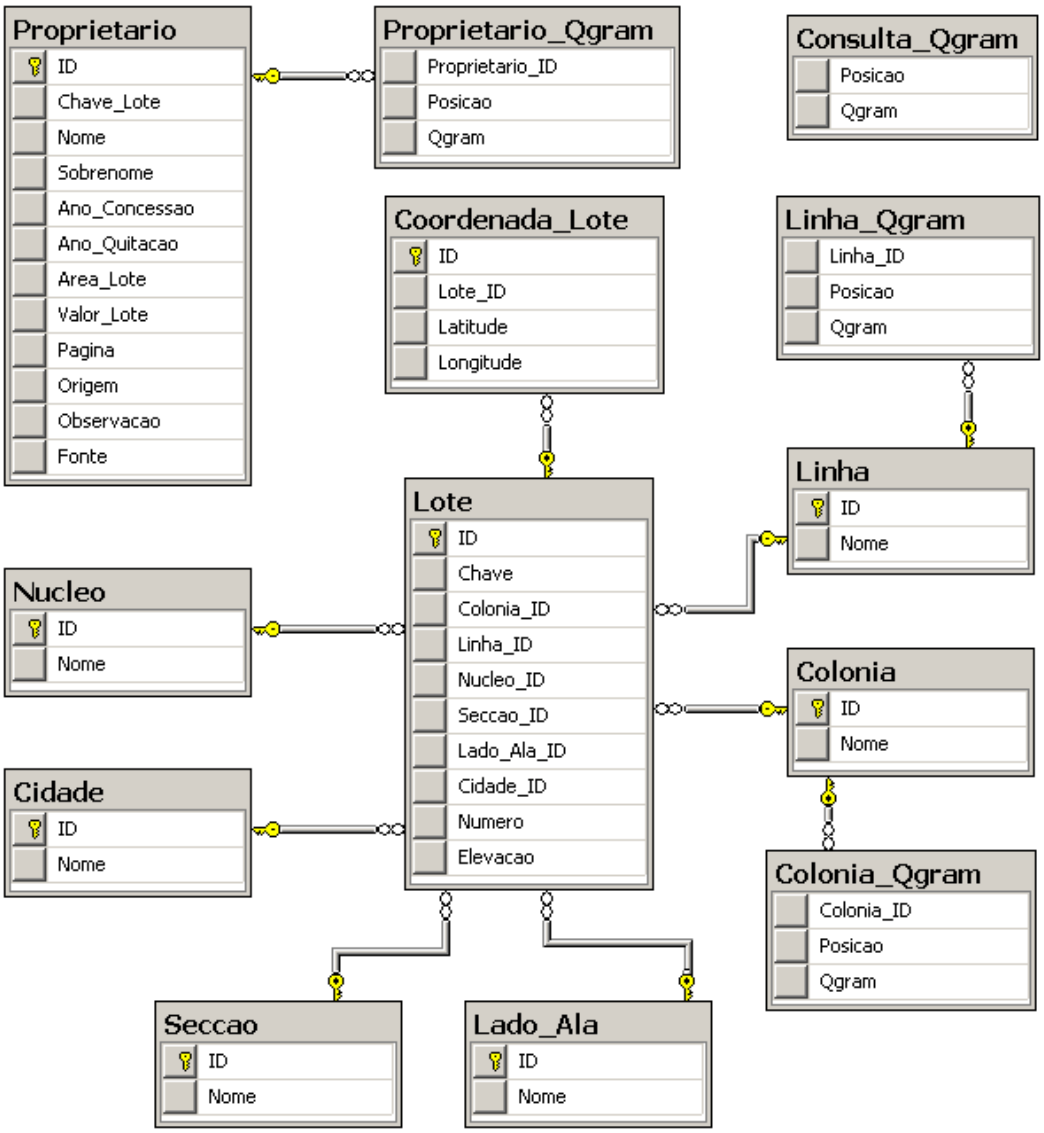


Figura 2.4: Modelo entidade relacionamento da aplicação.

É importante destacar, na Figura 2.4, como principal entidade a tabela de Lote, a qual possui relação de um para muitos com a tabela de coordenadas dos lotes.

As tabelas com sufixo *_Qgram* armazenam os dados referentes aos *q-grams* descritos no item 2.2 Busca de dados por similaridade.

### 2.7.5 Diagramas de classes

As classes foram organizadas em três diferentes tipos.

**Negócio** - Trata-se de uma categoria responsável por representar o modelo de dados armazenado no banco. Seu diagrama é mostrado na Figura 2.5.

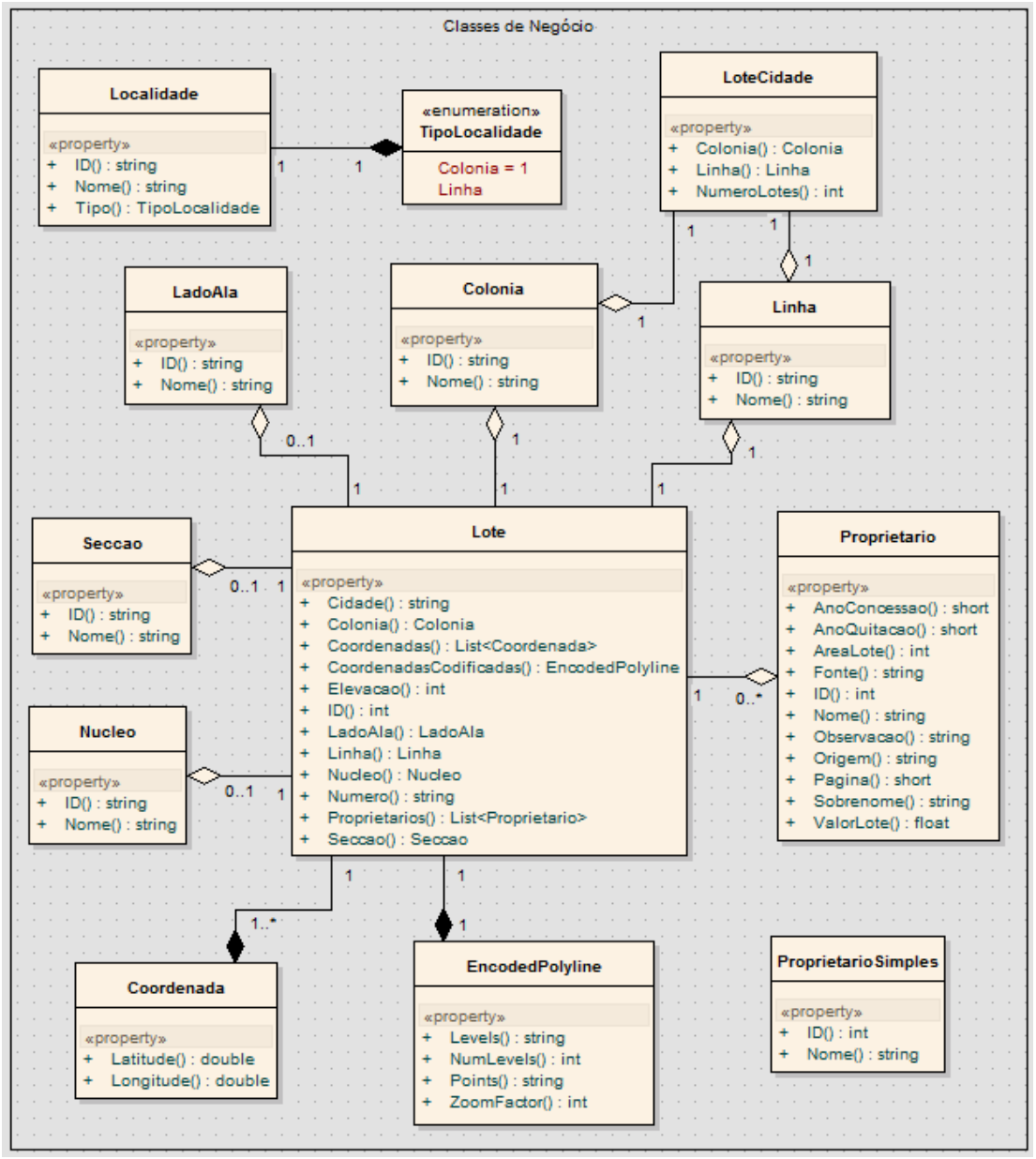


Figura 2.5: Diagrama de classes: classes de negócio.

**Dados** - Responsável por resgatar os dados do banco. Nesta categoria encontramos um classe responsável pelo CRUD de vários modelos com diversas SQL. Seu diagrama é mostrado na Figura 2.6.

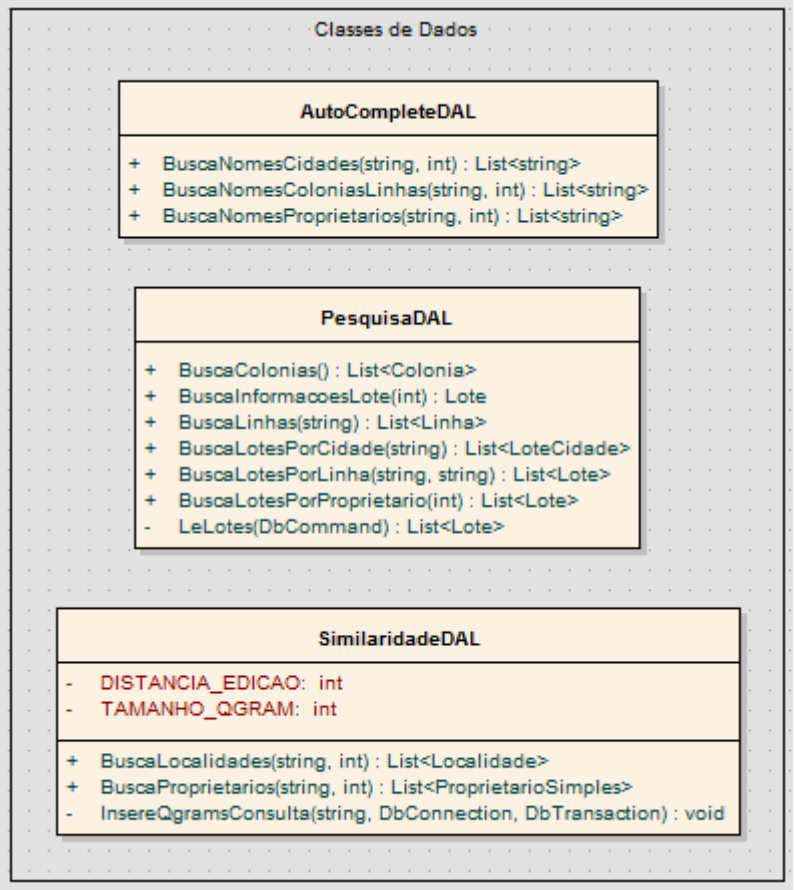


Figura 2.6: Diagrama de classes: classes de dados

**Úteis** - Trata-se de um categoria de classes que um framework mais robusto já deveria prover. Destaque para um classe para limpar dados que serão armazenados no banco de dados. Seu diagrama é mostrado na Figura 2.7.

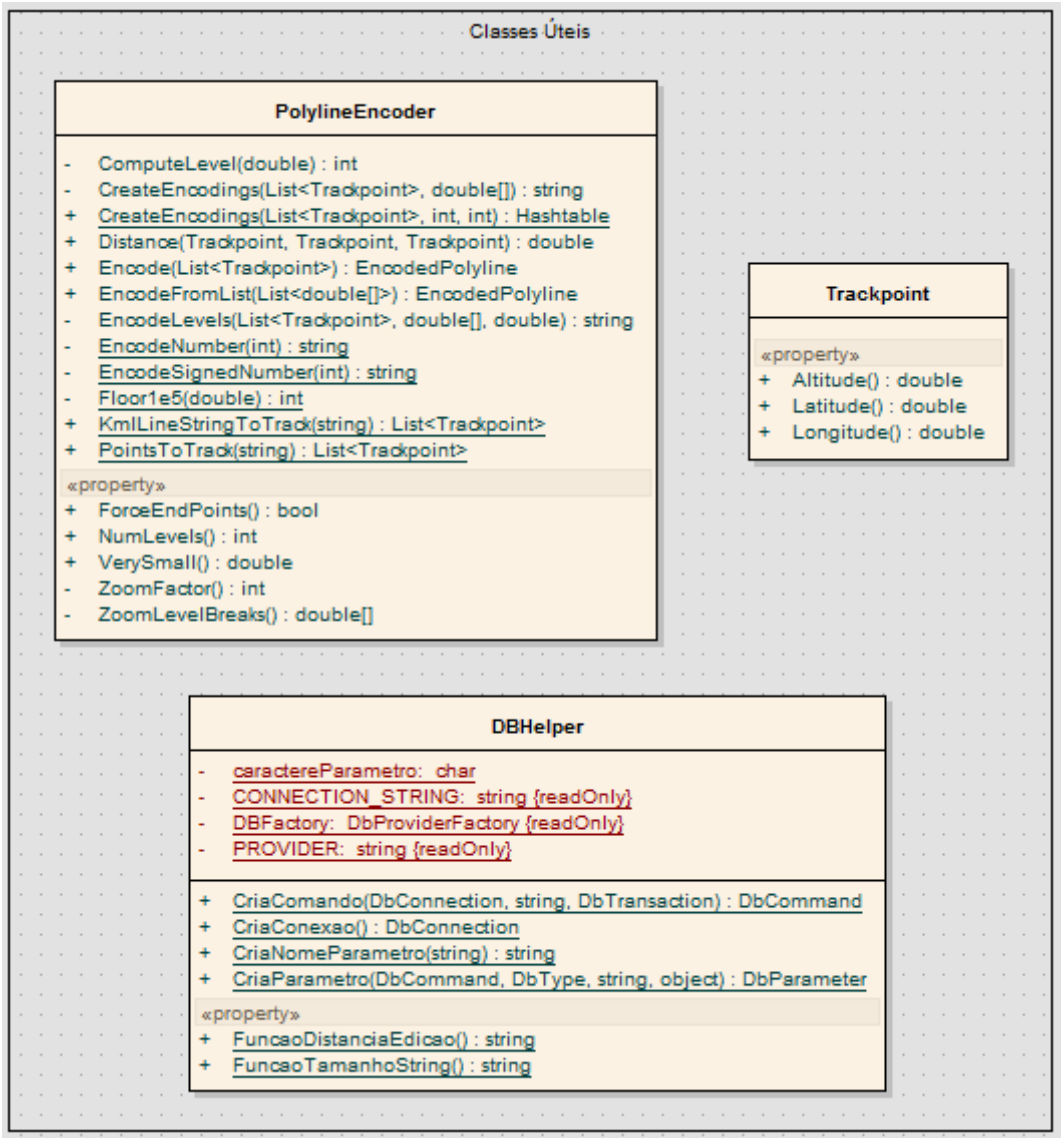


Figura 2.7: Diagrama de classes: classes úteis

Por fim, a arquitetura como um todo se demonstra bastante amarrada em nomes de tabelas e instruções SQL contendo classes com diversas linhas e função genérica.

### 2.7.6 Situação da aplicação

A aplicação Geo Colony versão 1.0 está em operação contemplando cerca de 21.000 lotes de colônias italianas e alemãs do Rio Grande do Sul. Conforme relatado no capítulo 3, tal aplicação necessita do licenciamento e uso de tecnologias proprietárias.

O Instituto de Informática da UFRGS, que mantém em operação a instância da aplicação, não possui infra estrutura de servidores com as devidas licenças requeridas. A necessidade exposta pelo Prof. Carlos Heuser, orientador deste trabalho, é a migração total da aplicação para tecnologias *open source* o que permitiria que a instância funcionasse em um servidor do Instituto e desfrutasse de backup e manutenção constante.

## 2.8 Ferramentas e tecnologias back end

### 2.8.1 RESTful

RESTful é uma API WEB implementada utilizando o protocolo HTTP e os princípios de REST. Cada requisição do cliente para o servidor deve indicar qual o método HTTP (GET, PUT, POST ou DELETE) e também o tipo de mídia aceito como retorno.

### 2.8.2 Hypertext Preprocessor (PHP)

Linguagem de *script server-side* arquitetada para a escrita de aplicações WEB. Seu código é interpretado e os comandos podem ser adicionados diretamente a um documento HTML. A linguagem foi criada em 1995 por Rasmus Lerdorf e possui tipagem dinâmica e fraca, orientação a objetos e funções de primeira ordem a partir da versão 5.2.

#### *2.8.2.1 Fuel PHP*

Trata-se de um framework HMVC simples, flexível e modular, desenvolvido pela comunidade em 2010 para PHP 5.3+. Sua documentação é bastante vasta e seu desenvolvimento é baseado nas melhores ideias encontradas em outros frameworks.

As principais funcionalidades é um sistema de rotas para a URL, implementação RESTful, validação dos dados, ORM e sistema de *cache*.

### 2.8.3 MySQL

Aclamado como o mais usado Sistema de Gerenciamento de Base de Dados (SGBD) relacional, o MySQL é praticado como escolha muito popular para a construção de aplicativos WEB. Como características podemos citar sua compatibilidade com a linguagem SQL:1999 e suporte para múltiplas plataformas.

## 2.9 Ferramentas e tecnologias front end

### 2.9.1 Hypertext Markup Language (HTML)

É a principal linguagem de marcação para criar páginas WEB e outras informações que podem ser mostradas em um WEB *browser*. HTML é escrito através de elementos que definem a estrutura da página, o visual dos componentes não é definido.

### 2.9.2 Cascading Style Sheets (CSS)

É uma linguagem de estilo utilizada para definir a apresentação de componentes dos documentos escritos em linguagens de marcação como o HTML. Sua função é definir tamanhos, margens, cores, tipo de fonte, dentre outros.

#### *2.9.2.1 Stylus*

Trata-se de uma linguagem da categoria dos pre-processadores. O Stylus introduz conceitos de variáveis, *mixins*, funções e condicionais. Sua função é prover uma maneira eficiente, dinâmica e muito mais expressiva para se gerar CSS.

### 2.9.3 Document Object Model (DOM)

É uma convenção *cross-plataform* e independente de linguagem para representar e interagir com objetos em documentos HTML. Os objetos são representados em formato de árvore e podem ser manipulados através de métodos através de JavaScript.

### 2.9.4 JavaScript

Trata-se de uma linguagem interpretada de *scripts* da WEB. Seu modelo é orientado a objetos por *prototype*, com funções de primeira ordem, tipagem dinâmica e fraca.

JavaScript foi desenvolvido pela Netscape por Brendan Eich em 1995 e se tornou um *standard* em 1996 sob o nome ECMAScript.

Originalmente desenvolvida para funcionar como parte dos *browsers*, permite a escrita de *scripts* que acessam e manipulam os objetos de uma página através da interface DOM além de orquestrar requisições ao servidor.

#### *2.9.4.1 CoffeeScript*

É uma pequena linguagem que transcompila para JavaScript. A linguagem adiciona *syntactic sugar* inspirado em Ruby, Python e Haskell para tornar o JavaScript mais conciso e de fácil leitura.

Seu código compila *um-por-um* para equivalente em JavaScript e as aplicações podem ser escritas com menos código sem afetar a performance.

#### *2.9.4.2 jQuery*

jQuery é uma biblioteca JavaScript que permite percorrer e manipular o DOM, tratar de eventos e fazer requisições ao servidor de uma maneira muito mais simples e com uma API que ultrapassa as diferenças de implementação entre os navegadores.

#### *2.9.4.3 Underscore*

Biblioteca JavaScript que prove uma série de mais de oitenta funcionalidades, sem estender os *built-in* objetos do JavaScript, inspiradas em funcionalidades já encontradas em linguagens como Ruby. Esta biblioteca é uma das dependências do framework Backbone.

#### *2.9.4.4 Backbone*

Sua função é estruturas aplicações WEB inspirado no padrão MVC. O framework prove uma maneira de representar os dados através de um *Model* que pode ser criado, validado, destruído e salvo no servidor.

Quando alguma interação altera o *Model*, este dispara um evento *change*. Assim, todas as *Views* que representam o estado deste são notificadas desta mudança podendo responder de mudando sua representação.

O framework foi escrito por Jeremy Ashkenas, que é autor do CoffeeScript e Underscore, em 2010 para operar em conjunto com aplicações RESTful via JSON.

### 2.9.5 XMLHttpRequest (XHR)

É uma API disponível na linguagem de programação JavaScript utilizada para enviar requisições HTTP diretamente para o servidor carregando a resposta, que pode ser no formato JSON, diretamente para o *script* de forma síncrona ou assíncrona.

O XHR possui um importante papel na técnica Ajax de desenvolvimento que consiste em se poder receber dados adicionais do servidor sem a necessidade de recarregar toda a página WEB no *browser*. Esta API é vastamente utilizada em aplicações modernas como Gmail, Facebook e o próprio Google Maps.

### 2.9.6 JavaScript Object Notation (JSON)

Trata-se de uma notação *open standard* de dados que possui boa leitura para humanos. É bastante utilizando como alternativa ao XML para o intercâmbio de dados em requisições à API de XHR. Sua especificação foi escrita por Douglas Crockford e é derivada das representações simples de objetos e *arrays* do JavaScript.

### 2.9.7 Web Storage

A especificação de *Web Storage*, certas vezes chamada de *Local Storage* ou DOM *Storage*, trata-se de uma API que possibilita que as páginas armazenem localmente informações através de uma mapa de chave e valor.

De acordo com (PILGRIM, 2013), seu funcionamento se assemelha com o dos *cookies*, tais informações persistem mesmo após deixar a página ou fechar o navegador. Porém, diferentemente dos *cookies*, estas não são transmitidas para o servidor. A vantagem do uso desta API está relacionado justamente a esta característica que permite o armazenamento de um volume maior de informações sem onerar em requisições.

# 3 MODELO DA APLICAÇÃO

A aplicação, denominada Geo Colony 2.0, por se tratar de uma migração, atende aos mesmos requisitos e casos de uso descritos em (DOS SANTOS, 2009).

O desenvolvimento segue o Manifesto Ágil (BECK, 2013) e portanto foi introduzido o conceito de Histórias de Usuário (do inglês *user stories*), que são narrativas que descrevem a interação do usuário com a aplicação. Este conceito possibilitou não só a redescoberta, organização e aprimoramento dos requisitos do sistema como também facilitou a priorização destes através de uma visão de relação de valor para o usuário *versus* complexidade de implementação.

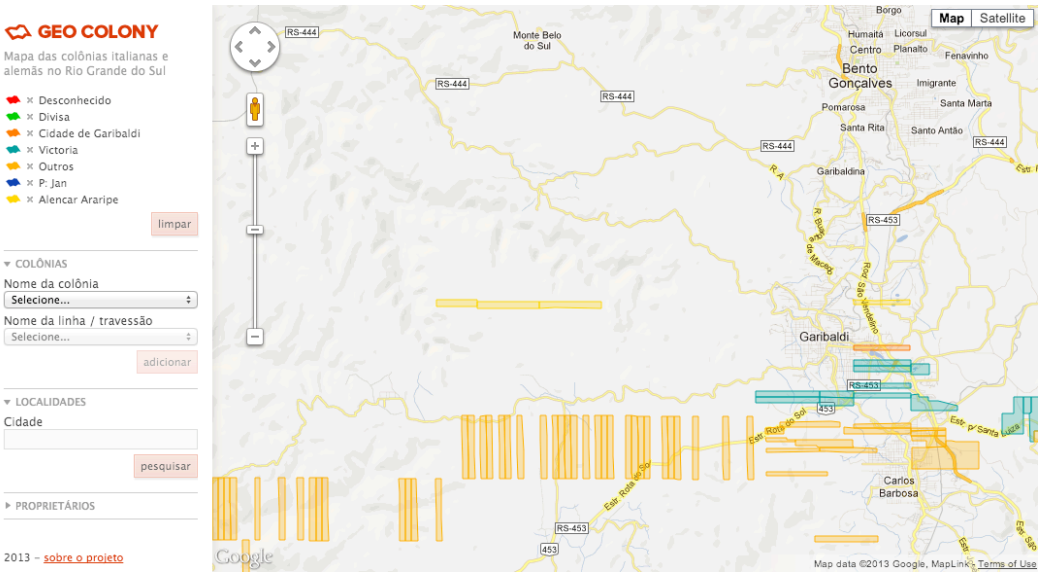


Figura 3.1: Tela da interface inicial da aplicação Geo Colony 2.0.

A Figura 3.1, na lateral esquerda, ilustra os territórios que estão sendo exibidos no mapa e três seções principais do menu da aplicação: pesquisa por colônias, pesquisa por localidades e pesquisa por proprietários. A porção central da interface constitui o mapa provido pela API do Google Maps com seus controles padrões a esquerda.

As funcionalidades da aplicação junto com suas respectivas Histórias de Usuário, em um modelo proposto por (NAZZARO e SUSCHECK, 2010), são descritas a seguir.

## 3.1 Painel de territórios

Tabela 3.1: Histórias de Usuário ligadas a funcionalidade de Painel de territórios

| *No papel de* | *Eu quero* | *Para* |
|---|---|---|
| Usuário | Poder ver todos os territórios mostrados no mapa | Não precisar procurá-los navegando pela interface. |
| Usuário | Que meus territórios fiquem armazenados na aplicação | Que não seja preciso busca-los novamente em um novo acesso. |
| Usuário | Poder remover um território | Reverter a ação de adicionar um território cometida por engano. |
| Usuário | Poder limpar a minha lista de territórios | Poder direcionar minhas buscas a outro ponto focal. |

A aplicação organiza uma lista dos territórios que estão sendo mostrados no mapa. Por território entende-se como os lotes de uma linha ou todos os lotes pertencentes a um proprietário que, na interface, é destacado através do prefixo "P".

O usuário pode destacar no mapa os lotes do território clicando em seu título. De outra maneira, através do símbolo em formato de "x", o território pode ser removido da lista. A lista é armazenada de forma persistente através da API de *Web Storage* no *browser* do cliente.


Figura 3.2: Painel de lista de territórios

## 3.2 Pesquisa por colônia, linha e localidade

Tabela 3.2: Histórias de Usuário ligadas a funcionalidade de Pesquisa por colônia e linha

| *No papel de* | *Eu quero* | *Para* |
|---|---|---|
| Usuário | Saber quais as linhas da colônia | Extrair esta informação. |
| Usuário | Poder adicionar à lista de territórios uma linha que pertence a determinada colônia | Poder visualizar esta linha no mapa. |
| Usuário | Poder pesquisar pelo nome de uma cidade | Poder listar as linhas e adiciona-las ao painel de territórios. |

A interface lista todas as colônias e, após selecionada alguma delas, suas linhas. A partir do momento em que uma linha é selecionada esta pode ser adicionada ao painel de territórios através do botão “adicionar”.

A busca por localidade inclui a correspondência entre os lotes de colonização e as atuais cidades que ocupam o mesmo território. O resultado da busca é uma lista de linhas que podem ser adicionadas ao painel de territórios. O conceito de busca por similaridade, introduzido na seção 2.6, é explorado.

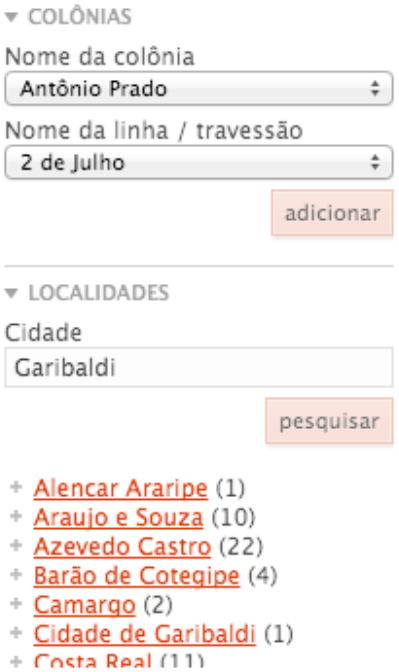


Figura 3.3: Pesquisa por colônia, linha e localidade

## 3.3 Pesquisa por proprietários

Tabela 3.3: Histórias de Usuário ligadas a funcionalidade de Pesquisa por proprietários

| *No papel de* | *Eu quero* | *Para* |
|---|---|---|
| Usuário | Poder pesquisar pelo nome de um proprietário | Poder adicionar à lista de territórios os seus lotes. |

A busca pelo nome do proprietário é possível através da interface. O resultado da busca é uma lista de linhas que podem ser adicionadas ao painel de territórios. O conceito de busca por similaridade, introduzido na seção 2.6, novamente é explorado.


Figura 3.4: Pesquisa por proprietários

## 3.4 Informações do lote

Tabela 3.4: Histórias de Usuário ligadas a funcionalidade de Informações do lote

| *No papel de* | *Eu quero* | *Para* |
|---|---|---|
| Usuário | Poder clicar nos lotes do mapa | Saber informações adicionais a seu respeito. |
| Usuário | Poder clicar nos lotes do mapa | Visualizar as informações de proprietários do lote. |

Ao clicar em um lote todas as informações de dados históricos, geoprocessamento e proprietários do lote são mostrados em uma janela. Os dados históricos são compostos pelo nome da colônia, nome da linha, número do lote, núcleo, secção, e lado/ala. Os dados de geoprocessamento são compostos pela área do lote, cidade atual mais próxima e elevação média aproximada. As informações do proprietário contemplam seu nome, família, ano de concessão, ano de quitação, valor do lote, entre outros.

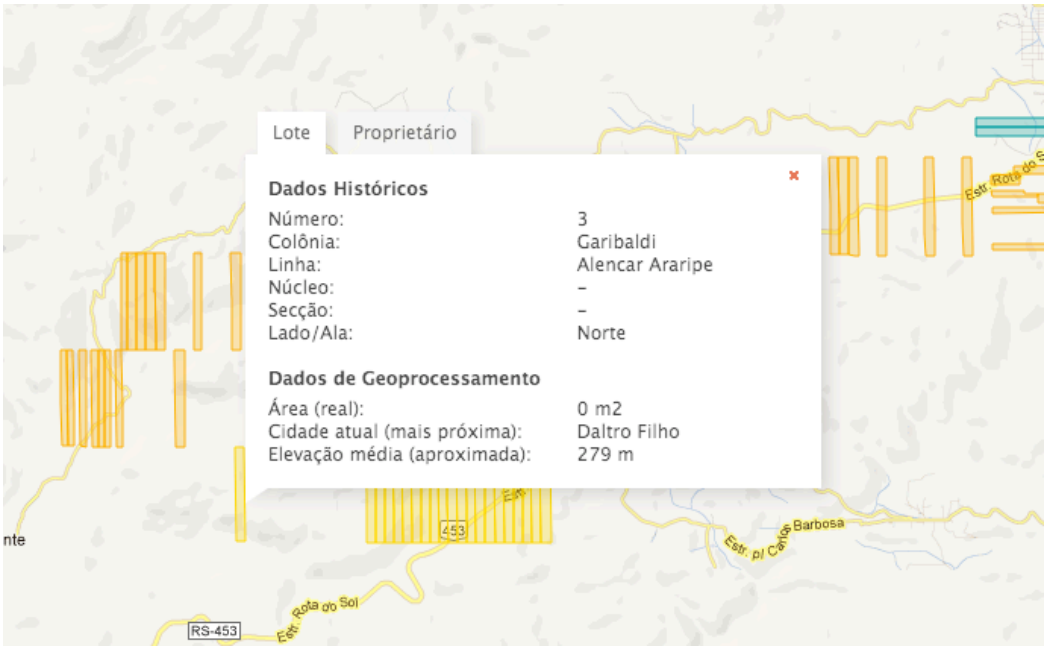


Figura 3.5: Informações do lote

# 4 IMPLEMENTAÇÃO DA APLICAÇÃO

O assunto deste capítulo será a implementação da aplicação com foco na separação das arquiteturas em *back end* e *front end*. Antes disto, a escolha de plataformas e metodologia de desenvolvimento serão apresentadas.

## 4.1 Escolha das plataformas

Antes de mais nada, é importante destacar a necessidade de escolha de tecnologias de código aberto e com suporte comum nos mais diversos ambientes de servidores.

A aplicação Geo Colony 1.0 fazia uso de ASP.NET e SQL Server no *back end*, para a migração, estas tecnologias precisaram ser substituídas. Segundo (ARAÚJO, 2009), a linguagem PHP possui melhor velocidade em relação ao ASP, é multi-plataforma (Linux, Solaris, Windows, por exemplo) e o mais importante para nossa aplicação: é de código fonte aberto. Estas vantagens justificam com solidez a escolha desta linguagem como sendo ideal para a aplicação. O SGBD substituto que se mostrou adequado, graças a seu excelente desempenho e estabilidade, foi o MySQL. PHP e MySQL juntos em um servidor constituem uma configuração bastante comum e, atestado pela minha experiência, nenhum desafio de integração entre ambos deverá ser enfrentado.

As tecnologias e linguagens do *front end* usadas na aplicação base já atendiam as necessidades expostas no modelo da aplicação. Um detalhe bastante importante observado durante a fase de planejamento, foi a total falta de uma organização e padrão da aplicação Geo Colony 1.0. Com isto em vista, a linguagem CoffeeScript e o *framework* Backbone foram escolhidos para fazer parte do desenvolvimento da nova aplicação. Juntos, eles permitiram a criação de diferentes estruturas e aplicaram *Design Patterns* importantes para assegurar a qualidade do código.

## 4.2 Reescrita da aplicação

Evoluções de ferramentas sempre são mais recomendadas em comparação a reescritas. A razão é que o processo pode ser feito de maneira gradual trazendo uma série de benefícios.

A aplicação Geo Colony 2.0, por consequência da adoção de novas linguagens, tecnologias e *design pattens*, inviabilizou ter seu código como o resultado de uma evolução da versão anterior. A arquitetura *front end* em especial, que não é afetada pela troca de linguagem de programação, é a base da interação da aplicação e a necessidade da adoção de um *framework* como o Backbone era essencial para a boa estruturação do seu código.

Em vista disto, a escolha pela total reescrita do código é mais acertada e constitui a única alternativa.

## 4.3 Metodologia de desenvolvimento

Conforme mencionado no capítulo três, a aplicação foi desenvolvida seguindo o Manifesto Ágil. O manifesto prega uma modelagem menos abrangente e foca mais na codificação da aplicação, além dos principais princípios de: entregas frequentes da aplicação em estado funcional, simplicidade e priorização de *software* em funcionamento mais que documentação abrangente.

Em sintonia com o manifesto, um repositório público foi criado no GitHub (http://github.com/jcemer/geo-colony) para manter o código da aplicação sob licença MIT. No momento de escrita deste texto, o repositório já possui mais de cem *commits* descrevendo cada nova entrega de funcionalidade ou correção de *bug* da aplicação.

## 4.4 Arquitetura geral e sua modelagem

A Figura 4.1 ilustra a arquitetura geral da aplicação que é baseada na comunicação entre cliente e servidor através da troca de mensagens com dados no formato JSON.

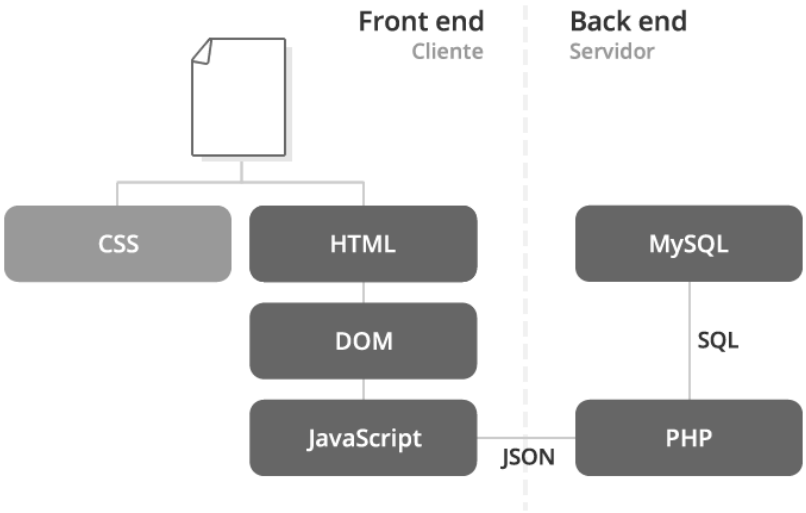


Figura 4.1: Arquitetura geral da aplicação

As duas faces seguem o padrão (H)MVC, de um lado através do *framework* Fuel PHP e do outro pelo *framework* Backbone. Assim como na aplicação base, toda a lógica é executada através do JavaScript no *browser* do cliente. A principal diferença é a presença do framework Backbone, apresentado na seção 2.9.4.4, que possibilita uma melhor estruturação do código seguindo um *design pattern*.

## 4.5 Arquitetura do back end da aplicação

A aplicação foi desenvolvida seguindo o padrão HMVC proposto pelo *framework* Fuel PHP. O código foi escrito seguindo as diretrizes de *Coding Standards* indicados na documentação do *framework* (http://fuelphp.com/docs/general/coding_standards.html).

O código da aplicação Geo Colony 1.0, por possuir uma diferente organização e estar escrito em outra linguagem conforme já mencionado, não foi utilizado.

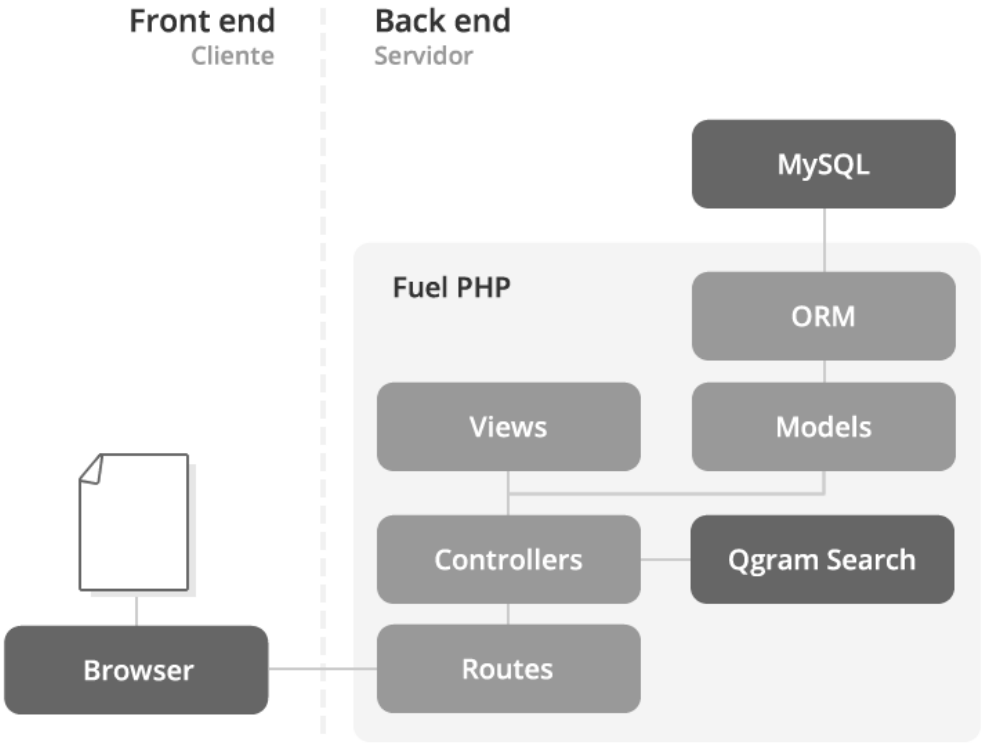


Figura 4.2: Arquitetura *back end* da aplicação

É importante destacar que o padrão HMVC implementado em *frameworks* WEB e em particular no Fuel PHP, possuem algumas divergências do padrão clássico proposto nos anos de 1980. A seguir serão apresentados todos diferentes componentes da arquitetura, mostrados na Figura 4.3, destacando suas características.

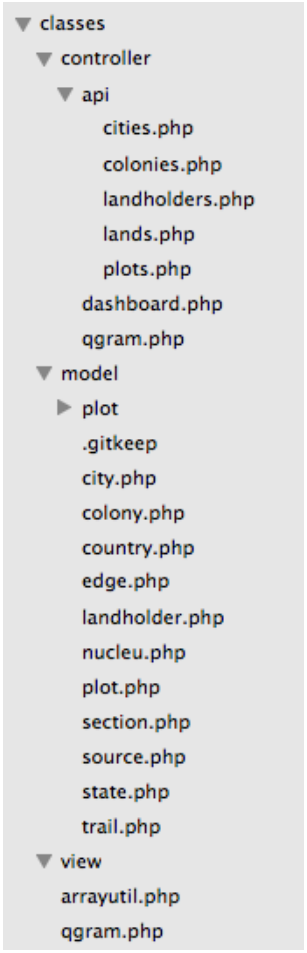


Figura 4.3: Estrutura de classe do *back end* da aplicação

### 4.5.1 Modelo de dados

A Figura 4.4 mostra a modelagem Entidade-Relacionamento do banco de dados da aplicação. Os nomes de tabela e suas colunas seguem a convenção adotada pelo *framework* utilizado.

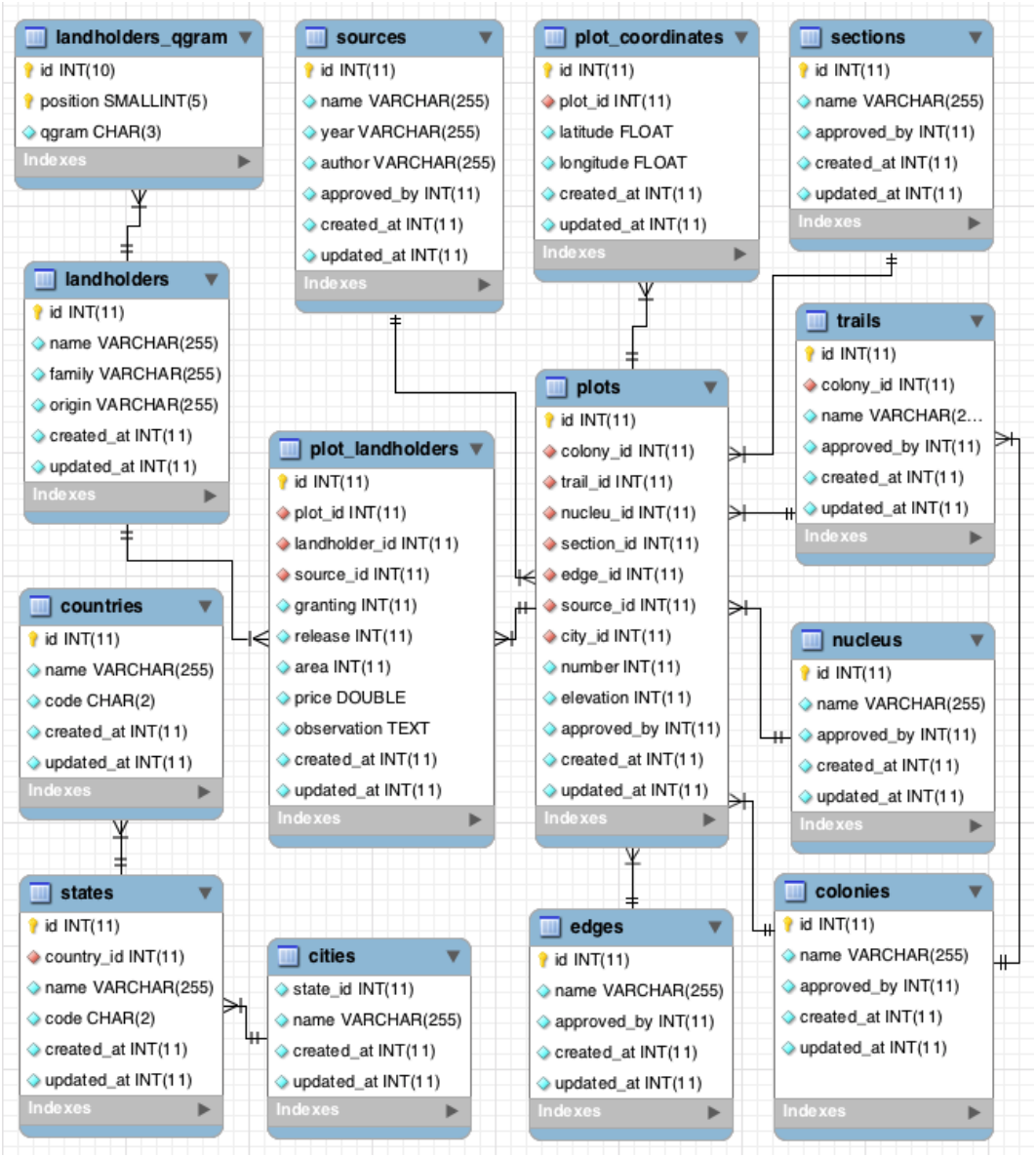


Figura 4.4: Modelo ER da aplicação

O principal dado da aplicação são os lotes, que possuem várias chaves para se conectar aos demais dados de: colônia, linha, núcleo, seção, lado/ala, fonte da informação e cidade atual. Um lote pode ter pertencido a vários proprietários que detêm diferentes informações sobre preço, área, data de compra e quitação.

O modelo abriga também os dados de coordenada pertencentes ao território demarcado por cada lote. Cada lote precisa possuir pelo menos três coordenadas e maioria deles possui muito mais que isto.

As tabelas com sufixo *_qgram* armazenam os dados utilizados pela busca por similaridade apresentada na seção 2.6.

#### *4.5.1.1 Migrations*

Projetos de aplicações de médio e grande porte merecem uma atenção especial quanto as alterações na estrutura do banco de dados. O *framework* utilizado, para auxiliar nestas tarefas, possui o conceito de *migrations*.

Através das *migrations* é possível criar arquivos sequenciais que registram quais as alterações feitas na estrutura do banco de dados para que, quando preciso fazer *deploy*, seja trivial replicar as mesmas mudanças aplicadas na plataforma de desenvolvimento. A aplicação Geo Colony 2.0 faz uso de *migrations* para a criação de todas as tabelas necessárias à base de dados.

### 4.5.2 Busca por similaridade (Qgram)

A busca por similaridade (DOS SANTOS, 2009 p. 15) foi implementada através de uma classe nomeada *Qgram*. Os métodos públicos e suas funcionalidades serão explicados a seguir.

**Generate** - Este método recebe um nome de tabela e opcionalmente o *id* do registro que se deseja gerar os *qgram* conforme ilustrado em (DOS SANTOS, 2009 p. 17). Sua função é popular as tabelas com sufixo _qgram para possibilitar as buscas futuras.

**Delete** - O método recebe um nome de tabela e opcionalmente o *id* do registro que se deseja remover seus *qgrams*. Sua função é remover os registros das tabelas com sufixo _qgram.

**Search** - Sua entrada é, mais uma vez, o nome da tabela e a *string* que se deseja buscar.

A mecânica da busca por similaridade é realizada através do *join* de duas tabelas do banco de dados, sendo uma delas com *qgrams* da *string* de busca e a seguinte a tabela com sufixo _qgram previamente populada.

A aplicação Geo Colony 1.0 utiliza uma tabela permanente no banco de dados para popular com os *qgrams* da *string* de busca, tal abordagem não permite buscas concorrentes. A implementação atual utiliza uma tabela temporária para cada conexão estabelecida com a base de dados evitando conflitos.

Uma segunda melhoria da aplicação é o uso de uma busca do tipo *like* com permutação das palavras que se deseja buscar. Associada a busca por similaridade, esta melhoria evita perdas de registros que, por diferença de quantidade de caracteres, são excluídos pela busca proposta por (GRAVANO, 2001).

### 4.5.3 Routes

A função das rotas (do inglês *routes*) é indicar ao componente *routing* como as URLs devem ser reconhecidas e qual a *action* de *controller* que deve ser disparada para atender determinada requisição.

A principal rota, ativada ao acessar a aplicação ou no *refresh* da página, é responsável por indicar a *action* responsável por retornar toda a interface da aplicação. Outras quatro rotas indicam a API responsável por retornar dados no formato JSON, estas são ativadas de maneira assíncrona durante a execução da aplicação para retornar informações de linhas, proprietários e lotes.

### 4.5.4 Controllers

Os *controllers* são classes que tem a função de, após de ser ativado por uma rota, retornar uma saída adequada. Convencionalmente em aplicações RESTful, caso da aplicação implementada, o *controller* é responsável por resgatar ou salvar os dados de um *model* e utilizar uma *view* para retornar HTML.

É importante destacar que os *controllers* com prefixo "API" são todos aqueles os quais suas *actions* estão adequadas a responder com dados no formato JSON sem consumir nenhuma *view*. A aplicação é composta pelos *controllers* apresentados a seguir juntamente com suas *actions*.

#### *4.5.4.1 Dashboard*

O *Controller Dashboard* é composto por uma única *action* ativada pela rota principal da aplicação. Sua função é resgatar todas as colônias através do *Model Colony* e renderizar a *View* responsável pela interface inicial da aplicação.

#### *4.5.4.2 API Cities*

Este *controller* é responsável pela pesquisa por localidades e possui as seguintes *actions*. (a) Search: É utilizada para gerar o *auto-complete* da interface de busca. (b) Search Trails: Retorna a lista de linhas que possuem correspondência com a cidade informada na busca.

#### *4.5.4.3 API Colonies*

Este *controller* possui apenas uma *action* responsável por retornar as linhas que pertencem a uma determinada colônia.

#### *4.5.4.4 API Landholders*

*Controller* que, através do método *search* da classe *Qgram*, retorna os proprietários dos lotes por busca de similaridade de nome.

#### *4.5.4.5 API Lands*

Este *controller* tem um papel bastante importante na aplicação, sua função é retornar a lista dos territórios mostrados na seção 3.1 de funcionalidade do painel de territórios. As duas *actions* são praticamente idênticas, com a diferença que uma delas retorna um único território e a outra é usada no *bootstrap* da aplicação no cliente, portanto retorna tantos territórios quanto o cliente já tiver selecionado.

O método auxiliar privado *factory* deste *controller* implementa o *Design Pattern* de mesmo nome. Sua função é analisar os parâmetros passados delegando ao *Model Trail* ou ao *Model Landholder*. Isto possibilita que o Painel de territórios mostre linhas e lotes de proprietários. Os dados retornados já incluem as coordenadas dos lotes para que sejam plotados no mapa assim que mostrados no painel.

#### *4.5.4.6 API Plots*

Trata-se do *controller* que possui uma única *action* responsável por retornar as informações mostradas na janela de mais informações dos lotes. Seus dados incluem aqueles descritos na seção 3.4.

### 4.5.5 Models

Em uma aplicação MVC, toda operação de recebimento, manipulação ou exclusão de dado deve ser feita através de um *model*. No Fuel PHP um *model* é uma classe mapeada diretamente a uma tabela do banco de dados e que, através de ORM, pode possuir relações do tipo *belongs to*, *has one*, *has many* e *many to many* com outros objetos.

Os *models* e suas relações permitem o acesso a dados sem que seja necessária a escrita de SQL e, nos casos mais complexos em que for preciso, as *queries* devem

sempre estar alocadas nos métodos dos *models*. Um exemplo é o método *findByNameWithPlots* do *Model City* que executa uma SQL específica responsável por procurar e retornar apenas as cidades que possuem pelo menos um lote.

Uma vantagem da arquitetura é a possibilidade de se ter *observers* que se responsabilizam em popular as colunas *created_at* e *updated_at* com a data de criação do e modificação do dado respectivamente.

### 4.5.6 Views

A aplicação é do tipo *single page* e portanto, possui apenas uma única *view* referenciada pelo *Controller Dashboard* conforme ilustrado na seção 4.5.4.1. Os demais *controllers* respondem em formato JSON e por isso não se faz necessário o uso de *views*.

## 4.6 Arquitetura do front end da aplicação

A aplicação desenvolvida, assim como a aplicação base, tem seu modelo de desenvolvimento centrado no lado cliente. Isto significa que a lógica de apresentação e de negócios da aplicação em grande parte é processada no *browser* do usuário por meio de JavaScript.

Segundo (MURPHEY, 2003), nos últimos tempos o desenvolvimento *front end* começou a ser levado mais a sério. Muitos novas ferramentas e técnicas surgiram e foram inseridas nesta nova aplicação. A aplicação base não possuía nenhum tipo de organização no seu código JavaScript, como veremos a seguir, Backbone foi o grande responsável pela mudança da organização e da maneira de pensar no fluxo de dados e lógica da aplicação.

Todo o código do comportamento da aplicação foi escrito na linguagem CoffeeScript, apresentada na seção 2.9.4.1. A linguagem utilizada para apresentação na aplicação foi o Stylus, apresentado na seção 2.9.2.2. As linguagens utilizadas são compiladas respectivamente para JavaScript e CSS.

### 4.6.1 Tarefa de build

As linguagens utilizadas para gerar os arquivos de JavaScript e CSS, chamados de *assets*, necessitam ser compiladas através do Node.js. A tarefa de *build* constitui chamadas as APIs destes compiladores, escritos em JavaScript, com as indicações de quais arquivos devem ser compilados para um determinado arquivo destino.

Esta tarefa deve ser executada através de um terminal de linha de comando e é imprescindível que o Node.js e seu gerenciador de pacotes NPM esteja instalado. Os comandos *npm install* (faz o download dos compiladores) e *cake build* disparam o *build* simples dos *assets* gerando arquivos sem espaços não necessários ou comentários para diminuir seu tamanho em Kilobyte.

Com o intuíto de facilitar o desenvolvimento, é possível adicionar o comando *dev* para que os arquivos sejam gerados com comentários e espaços, facilitando a leitura e identificação de possível erros. Uma segunda tarefa, executada pelo comando *cake watch*, ativa um processo que assiste a modificações nos arquivos e recompila os *assets* a cada nova alteração.

### 4.6.2 Comportamento

O comportamento da aplicação, apresentado na Figura 4.5 e codificado em CoffeeScript é apoiado no *framework* Backbone. O Backbone foi desenvolvido para ser utilizado como um aplicação escrita em Ruby on Rails, famoso *framework* MVC em Ruby e, conforme ilustrado na seção 2.9.4.4, introduz uma variação do padrão MVC ao *front end* das aplicações.

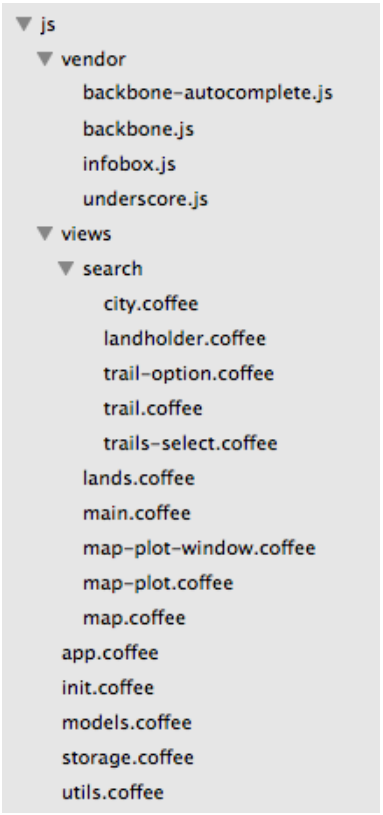


Figura 4.5: Arquivos da arquitetura *front end*

A Figura 4.6 ilustra a arquitetura e as ligações de dependências entre as diferentes partes e tecnologias. Diferente do modelo MVC tradicional, o conceito de *model* é dividido em *collection* e *model*. A *collection* pode referenciar uma URL que do servidor que possa retornar mais de um registro (*model*) e para servir como agrupador de *models* possibilitando que sejam representados em conjunto em uma *view*.

Tanto *collections* quanto *models* podem sincronizar o estado de seus dados com o servidor e notificar, através de eventos, as *views* das respectivas mudanças.

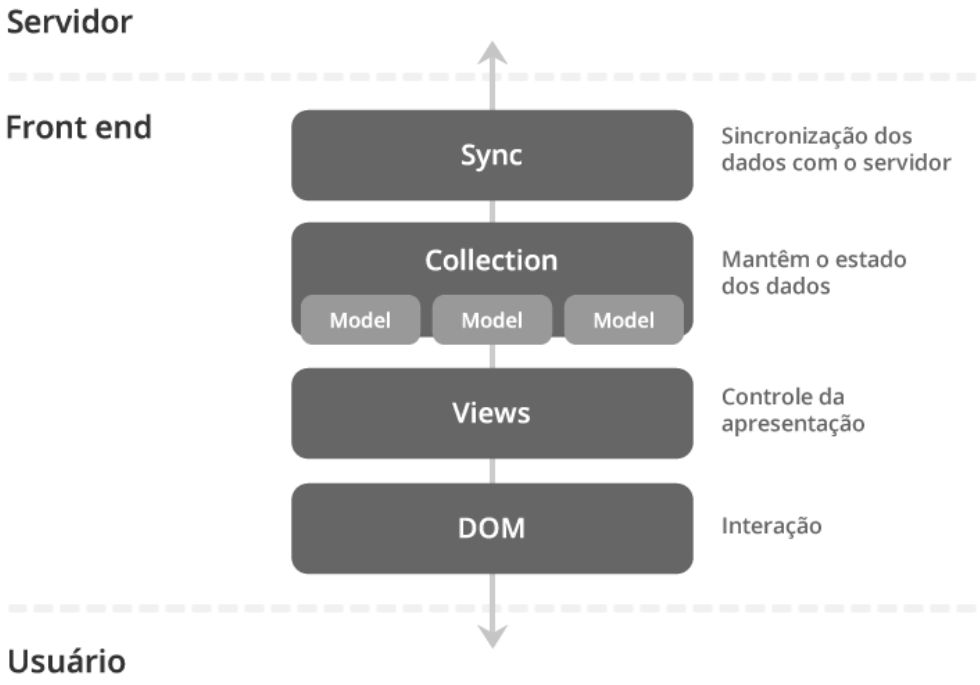


Figura 4.6: Arquitetura do *front end* implementada pelo Backbone

A seguir serão apresentadas todas as estruturas criadas na aplicação com base na organização proposta pelo Backbone.

*4.6.2.1 Models e Collections*

As declarações de *models* são bem mais simples se comparadas com as que precisamos escrever no *back end* da aplicação. Outro detalhe importante, é que apenas os *models* utilizados diretamente pelas *views* é que serão criados e então não teremos necessariamente os mesmos *models* do *back end*.

Os *models* e suas *collections* serão apresentados a seguir. Por convenção, o nome do *model* estará no singular e o da sua *collection* no plural. É importante observar que aqueles nomeados com o prefixo "search" fazem parte da seção lateral da interface utilizada para buscar os territórios, estes se comunicam com *actions* do *back end* que retornam poucos dados satélites dos registros e, portanto, necessitam desta distinção.

**Land e Lands** - Estes são responsáveis por coletar os dados do disponibilizados pelo *controller* do *back end* mostrado na seção 4.5.4.5.

**Plot e Plots** - Responsáveis pelas informações dos lotes, são correspondentes do *controller* apresentado na seção 4.5.4.6. Assim como mencionado na seção 4.5.4.5, o *Model Land* é retornado com as informações de coordenada do lote. A requisição ao *controller* do *back end* apenas é necessária caso seja necessárias informações adicionais. Neste caso, a requisição constitui uma alteração no estado dos dados do modelo.

**Search Trail e Search Trails** - Responsáveis pelos dados de linhas. Sua correspondência é o *controller* apresentado em 4.5.4.3 com alteração dinâmica da URL de sincronismo para adição da informação de qual colônia a coleção pertence.

**Search City e Search Cities** - Estes são responsáveis por coletar os dados do disponibilizados pelo *controller* do *back end* mostrado na seção 4.5.4.2.

**Search Landholder e Search Landholders** - Responsáveis pelas informações dos proprietários, são correspondentes do *controller* apresentado na seção 4.5.4.4.

*4.6.2.2 Views*

A arquitetura proposta pelo *framework* não possui um *controller* e, por sua vez, as *views* assumem também este papel sendo responsáveis por abrigar os *models* e *collections*, monitorar eventos do DOM; além de serem responsáveis pela apresentação.

A aplicação é gerenciada por uma única *view* principal, *View Main*, responsável por gerenciar a *Collection Land* armazenando na *Storage* (seção 4.6.2.4) os identificadores dos *models* que a constituem. A cada novo território adicionado, um *Model* é instanciado e adicionado a *collection*. No *bootstrap* da aplicação uma consulta a *Storage* garante a instanciação dos territórios selecionados em uma visita passada.

A *View Main* também é responsável por instanciar as demais *views* da aplicação que são organizadas em três categorias: territórios, busca e mapa.

A categoria de territórios é composta por uma única *view* que renderiza o painel dos territórios já selecionados pelo usuário. Esta *view* espera eventos de *add, change* e *remove* associados a *collection* de *Lands*, que também é compartilhada por outras *views*, com a função de alterar a representação do painel conforme diferentes interações do usuário.

A *View City*, pertencente a categoria de busca, apresenta o painel de busca de cidades. As *collections* de *Search Cities* e *Search Trails* são utilizadas, a primeira faz parte de um *subview* de auto-complete do campo de busca por cidade, a segunda retorna os resultados da busca. A categoria de busca possui mais outras duas *views*: *trail* e *landholders* com funcionamento análogo ao da *View City*. Um destaque apenas para a *View Trail*, que possui mais duas *subviews* responsáveis por renderizar o campo de seleção de linhas.

Por sua importância para a aplicação, cada uma das *views* da categoria mapa, que dependem entre si, serão apresentadas a seguir.

**Map** - Responsável por instanciar o objeto *Map* da API do Google Maps, calcular os limites dos territórios, aplicar *zoom* a um determinado território e assistir aos eventos de *change*, *reset* e *remove* da *collection* de territórios. Esta *view* é quem cria as *views* do tipo *Map Plot* descritas a seguir.

**Map Plot** - Um território é composto por um ou mais lotes, esta *view* recebe um *Model Plot* e o renderiza através da instanciação de um objeto do tipo *Polygon* da API do Google Maps. Apenas as coordenadas no formato aceito pela API de mapas são necessárias para a renderização. Um evento do tipo *on click* é associado para invocar a criação de uma *view* do tipo *Map Plot Window*.

**Map Plot Window** - Última das *views*, é responsável por renderizar a janela com as informações do lote. Além disto, é necessário que ela assista alterações ao *model* de lote vinculado para que possa atualizar suas informações. O controle da navegação entre informações gerais e proprietários também é uma de suas tarefas.

*4.6.2.3 Google Maps Info Window*

A aplicação base utilizava a API do Google Maps para apresentar as informações dos lotes através de uma janela. Por consequência da quantidade de informação, abas de conteúdo foram criadas para separar dados do lote de seus proprietários.

A nova API 3.0 do Google Maps não possui mais suporte para o uso de abas de conteúdo. Em virtude disto, a biblioteca Info Box documentada em (LITTLE, 2010) foi utilizada. Através dela foi possível a criação das abas e uma melhor estilização da interface, conforme apresentado na Figura 3.5.

*4.6.2.4 Storage*

A tecnologia de API de Web Storage, apresentada na seção 2.9.7, permite o armazenamento de informações de maneira perene no *browser* do cliente.

As informações são armazenadas através de um mapa de chave e valor, sendo o valor necessariamente um objeto do tipo *string*. Considerando esta limitação, foi necessário o desenvolvimento de uma classe com a finalidade de prover a API apresentada a seguir, que recebe as informações de *ids* dos territórios e as armazena em um formato compatível.

O objeto criado através da classe deve informar um identificador único, em relação ao domínio, utilizado para armazenar as informações na API de Web Storage.

**Load** - Este método já é disparado no momento da criação de um objeto da classe, sua função é resgatar os valores previamente armazenados.

**Save** - Este método coleta a lista de ítem, realiza sua conversão para *string* e armazena o resultado através da API de Web Storage.

**Itens -** Constitue um atributo do objeto que contém a lista dos itens.

**Add** - Adiciona um novo valor a *storage*, este método automaticamente valida se o valor já não está presente para evitar duplicadas e chama o método *save*, apresentado a anteriormente.

**Remove** - Este método remove um valor específico da lista de itens. Assim como o método *add*, invoca o método *save*.

**Reset** - A utilidade deste método é a completa destruição dos ítens e de sua representação armazenada através da API de Web Storage.

### 4.6.3 Apresentação

A apresentação da aplicação é escrita na linguagem Stylus que posteriormente é compilada para um único arquivo do tipo CSS que indica o estilo de cada um dos elementos presentes no documento HTML.

Com o intuito de permitir uma apresentação homogenia nos diferentes *browsers*, uma série de regras definidas em (MEYER, 2011) foram incluídas. Com a mesma intenção, uma biblioteca chamada Nib foi incluída para facilitar a escrita de regras que possuem diferentes sintaxes de acordo com a versão e fabricante do *browser* utilizado.

# 5 MIGRAÇÃO DOS DADOS

Migração de dados sempre constitui uma tarefa ardilosa e que exige apoio de ferramentas específicas. Em nosso caso, é necessário portar uma base de dados armazenada em SQL Server para o MySQL. A seguir serão apresentados os *softwares* utilizados e o procedimento adotado.

## 5.1 Aplicativo SQLyog

SQLyog é um aplicativo poderoso para gerência de bancos de dados MySQL. Dentre suas principais funcionalidades está o *Query Profiler, Visual Query Builder* e *Schema and Data Sync*.

O aplicativo disponibiliza uma ferramenta para importação de dados externos. Através de *Open Database Connectivity* (ODBC), API padrão para acessar diferentes SGBDs, é possível estabelecer uma conexão com um banco SQL Server. Após a seleção da *database*, é possível exportar as tabelas de dados.

Neste trabalho, a versão 11.1 de avaliação do aplicativo foi utilizada. Para a importação de dados, nesta versão, a restrição imposta é a de que apenas duas tabelas podem ser importadas por vez. Este limitante torna a operação mais trabalhosa mas não impede que o processo seja feito por completo.

## 5.2 Validando a importação

Nesta etapa, muitas validações empíricas foram aplicadas. A checagem de uma amostragem talvez seja a mais comum delas.

Uma abordagem importante neste processo é a conferência da quantidade de linhas em cada uma das tabelas. A instrução para SQL Sever, Figura 5.1, retirada de (SINDOL, 2011) retorna um relatório com a quantidade de todas as linhas das tabelas da base de dados.

```
DECLARE @QueryString NVARCHAR(MAX) ;
SELECT @QueryString = COALESCE(@QueryString + ' UNION ALL ','')
                    + 'SELECT '
                    + '''' + QUOTENAME(SCHEMA_NAME(sOBJ.schema_id))
                    + '.' + QUOTENAME(sOBJ.name) + '''' + ' AS [TableName]
                    , COUNT(*) AS [RowCount] FROM '
                    + QUOTENAME(SCHEMA_NAME(sOBJ.schema_id))
                    + '.' + QUOTENAME(sOBJ.name) + ' WITH (NOLOCK) '
FROM sys.objects AS sOBJ
WHERE
      sOBJ.type = 'U'
      AND sOBJ.is_ms_shipped = 0x0
ORDER BY SCHEMA_NAME(sOBJ.schema_id), sOBJ.name ;
EXEC sp_executesql @QueryString
GO
```

Figura 5.1: Instrução SQL Sever que retorna a quantidade de linhas

A instrução da Figura 5.2 permite contar a quantidade de linhas de uma tabela em MySQL. A instrução precisa ser executada informando as diferentes tabelas que se deseja consultar.

```
SELECT COUNT(*) FROM <table>
```

Figura 5.2: Instrução MySQL que retorna a quantidade de linhas

# 6 CONCLUSÃO

Este trabalho apresentou um estudo de caso de migração de uma aplicação que utilizava linguagens e *framework* de código fechado para soluções de código aberto. O processo todo foi calcado em *Design Pattenrs* e *Code Standards* destacando-se a importância do uso destes dois conceitos.

Assim como na versão 1.0, o resultado da migração faz uso do Google Maps através de sua mais nova API versão 3.0 para JavaScript. A partir dela, foi oferecido ao usuário a interação com os lotes coloniais através de sua plotagem em uma cartografia atual.

A aplicação foi codificada através de métodos ágeis com base em Histórias de Usuário. Uma série de ferramentas de código aberto foram utilizadas, em especial no *front end* da aplicação, para privilegiar a organização e facilitar futuras evoluções e manutenções da aplicação.

## 6.1 Trabalhos futuros

O trabalho desenvolvido por (DOS SANTOS, 2009), já destacava uma série de ideias que poderiam aprimorar a experiência do usuário, dentre elas possibilitar que os usuários mantivessem as informações a respeito dos lotes.

Seguindo e aprimorando estas ideias, o trabalho implementado por (LIMA, 2010), propõe uma infraestrutura genérica e escalável para o gerenciamento da reputação de usuários aplicado ao sistema de busca e exibição de dados georreferenciados.

Por fim, o trabalho de (FOLLE, 2012), permite a edição de áreas georreferenciads no Google Maps, permitindo a alteração da posição e tamanho dos lotes.

Estes dois trabalhos não fizeram parte do escopo desta migração, apesar de sua importância para constituir uma aplicação completa. Ao mesmo tempo, estes trabalhos tornariam a aplicação auto sustentável pela possibilidade de inserção de novos lotes e alteração das informações já presentes.

O resultado deste trabalho, apoiado na metodologia de desenvolvimento adotada e nos demais *frameworks* e tecnologias, já prevê o acoplamento destas evoluções de maneira trivial. O que pode gerar futuros trabalhos baseados no aprimoramento e implementação destas funcionalidades.

# REFERÊNCIAS


DOS SANTOS, Vinicius Rosa. **Sistema de busca e exibição de dados georreferenciados**. Novembro de 2009.

SCOTT, Hanselman. **Building Web Apps with ASP.NET**. Disponível em: http://www.hanselman.com/blog/BuildingWebAppsWithASPNETJumpStart8HoursOfFREETrainingVideos.aspx. Acesso em: Maio de 2013.

SQL SERVER. Disponível em: http://www.microsoft.com/en-us/sqlserver/default.aspx. Acesso em: Maio 2013

PHP. Disponível em: http://php.net. Acesso em: Maio de 2013

MYSQL. Disponível em: http://www.mysql.com. Acesso em: Maio de 2013

KHAN, Tamuir. **Getting Started with Stylus - CSS Pre-Processor**. Disponível em: http://bootstrap.pk/tutorials/getting-started-with-stylus-css-pre-processor. Acesso em: Maio de 2013.

COFFEESCRIPT. Disponível em: http://coffeescript.org. Acesso em: Maio de 2013.

MOZILLA. **JavaScript Overview**. Disponível em: https://developer.mozilla.org/en-US/docs/JavaScript/Guide/JavaScript_Overview. Acesso em: Maio de 2013.

NODEJS. Disponível em: http://nodejs.org. Acesso em: Maio de 2013.

GOOGLE. Disponível em: http://maps.google.com. Acesso em: Abril de 2013.

GIT. Disponível em: http://git-scm.com. Acesso em: Maio de 2013.

GRAVANO, Luis; et. al. **Approximate String Joins in a Database (Almost) for Free.** Proceedings of the 27th International Conference on Very Large Data Bases. San Francisco, CA, EUA. ACM 2001. p. 491-500.

SOMMERVILLE, Ian. **Software engineering**. 9.ed. Boston: Addison-Wesley, 2009.

BUSCHMANN, Frank. **Pattern-Oriented Software Architecture**. Disponível em: http://st-www.cs.illinois.edu/users/smarch/st-docs/mvc.html. Acesso em: Abril de 2013.

PILGRIM, Mark. **Dive Into HTML5**. Disponível em: http://diveintohtml5.info. Acesso em: Abril de 2013.

NAZZARO, William F., e SUSCHECK, Charles. **New to user stories?**. Disponível em: http://www.scrumalliance.org/articles/169-new-to-user-stories. Acesso em: Abril de 2013.

BECK, Kent. et al. **Manifesto for Agile Software Development**. Disponível em: http://www.agilemanifesto.org. Acesso em: Abril de 2013.

ARAÚJO, Fabrício. **Vantagens e desvantagens do PHP**. Disponível em: http://www.inforlogia.com/vantagens-e-desvantagens-do-php. Acesso em: Abril de 2013.

MURPHEY, Rebecca. **A Baseline for Front-End Developers**. Disponível em: http://rmurphey.com/blog/2012/04/12/a-baseline-for-front-end-developers. Acesso em: Abril de 2013.

LITTLE, Gary. **InfoBox**. Disponível em http://google-maps-utility-library-v3.googlecode.com/svn/trunk/infobox/docs/reference.html. Acesso em Abril de 2013.

MEYER, Kathryn. **CSS Tools: Reset CSS**. Disponível em: http://meyerweb.com/eric/tools/css/reset. Acesso em Abril de 2013.

SINDOL, 2011. **SQL Sever Row Count for all Tables in a Database**. Disponível em http://www.mssqltips.com/sqlservertip/2537/sql-server-row-count-for-all-tables-in-a-database. Acesso em Abril de 2013.

LIMA, Douglas de Oliveira. **Infra-estrutura para gerenciamento de reputação de usuários e sua aplicação em um caso real**. Novembro de 2010.

FOLLE, Priscila Azevedo. **Editando áreas georreferenciadas no Google Maps**. Julho de 2012.