

UNIVERSIDADE ESTADUAL DE MONTES CLAROS  
Centro de Ciências Exatas e Tecnológicas  
Departamento de Ciências da Computação  
Curso de Sistemas de Informação

Rafael Chagas Barbosa

DESENVOLVIMENTO DE APLICAÇÕES GEOGRÁFICAS  
UTILIZANDO O DJANGO/GEODJANGO

Montes Claros – MG  
Junho/2012

**Rafael Chagas Barbosa**

**DESENVOLVIMENTO DE APLICAÇÕES GEOGRÁFICAS  
UTILIZANDO O DJANGO/GEODJANGO**

**Monografia apresentada ao Curso de  
Sistemas de Informação, da Universidade  
Estadual de Montes Claros, como exigência  
para obtenção do grau de Bacharel em  
Sistemas de Informação.**

**Orientador: PROFESSOR ESPECIALISTA  
EDUARDO DINIZ AMARAL.**

**Montes Claros - MG  
Junho/2012**

**Rafael Chagas Barbosa**

**DESENVOLVIMENTO DE APLICAÇÕES GEOGRÁFICAS  
UTILIZANDO O DJANGO/GEODJANGO**

**Monografia apresentada ao Curso de  
Sistemas de Informação, da Universidade  
Estadual de Montes Claros, como exigência  
para obtenção do grau de Bacharel em  
Sistemas de Informação.**

**Orientador: PROFESOR ESPECIALISTA EDUARDO DINIZ AMARAL**

**Membros:**

---

**Prof. Msc. Wandré Nunes Pinho Veloso**

---

**Prof. Msc. Patrícia Takaki Neves**

**Montes Claros - MG  
Junho/2012**

## **DEDICATÓRIA**

*Dedico este trabalho a minha mãe, que sempre me apoiou nos momentos mais difíceis e sempre priorizou a minha educação.*

## **AGRADECIMENTOS**

Agradeço a Deus, pelas bênçãos e forças. A minha mãe por todo o amor e carinho; à minha namorada, Larissa, pela paciência e companheirismo; as minhas irmãs, familiares, colegas e amigos que de alguma contribuíram para a conclusão desse trabalho. Um agradecimento especial às colegas de turma Rhayane Stéphane e Rozileni Vieira que compartilharam comigo todos os momentos dessa longa caminhada.

## RESUMO

A utilização de aplicações geográficas na *web* está se tornando cada vez mais comum devido à, principalmente, popularização dos dispositivos móveis com acesso a internet. Com a demanda crescente por esse tipo de aplicação, os desenvolvedores perceberam a necessidade de novas ferramentas e metodologias que os ajudassem no desenvolvimento de aplicações em períodos curtos e que garantissem a mesma qualidade e segurança. O presente trabalho tem a finalidade de realizar um estudo sobre o desenvolvimento ágil de aplicações geográficas utilizando o *framework* Django e suas convenções. Além de reunir em um único documento as orientações necessárias e suficientes para se desenvolver uma aplicação. Para demonstrar a viabilidade de desenvolver aplicações geográficas utilizando o Django, foi desenvolvido um protótipo que teve como objetivo demonstrar a integração das ferramentas e tecnologias abordadas no trabalho.

**Palavras-chave:** aplicações geográficas, desenvolvimento web, Django, *framework*.

## **ABSTRACT**

The utilization of geographical applications in the web is becoming more common, due to the popularization of the mobile devices with access to internet. With the growing demand by that kind of application, the developers perceived the necessity of new tools and methodologies that helped them in the development of application in short periods with the same quality and security. The present work has the purpose of carry out a study about the agile development of geographical application using the framework Django and its conventions. In addition to cluster in a single document the guidelines necessary and sufficient to develop an application. To demonstrate the feasibility of develop geographical application using the Django, was developed a prototype that had as objective demonstrate the integration of the tools and technologies approached in the work.

**Keywords:** geographical applications, development, web, Django, framework.

## LISTA DE FIGURAS

Figura 1 - Funcionamento padrão de uma aplicação web .....	15
Figura 2 - Arquitetura de sistemas de informação geográfica.....	16
Figura 3 - Ilustração da interação entre as camadas da arquitetura MVC.....	18
Figura 4 - Exemplo de declaração de Javascript .....	20
Figura 5 - Exemplo de tipagem dinâmica.....	28
Figura 6 - Exemplo de números inteiros em Python .....	30
Figura 7 - Exemplos de números de ponto flutuante em Python .....	30
Figura 8 - Exemplos de números complexos em Python .....	31
Figura 9 - Exemplos de valores booleanos.....	32
Figura 10 - Exemplos de strings em Python.....	33
Figura 11 - Exemplos de lista em Python.....	34
Figura 12 - Exemplo de tuplas em Python .....	35
Figura 13 - Exemplos de dicionários em Python.....	36
Figura 14 - Sintaxe do comando <i>if</i> em Python .....	38
Figura 15 - Sintaxe do comando <i>while</i> em Python.....	39
Figura 16 - Sintaxe do comando <i>for</i> em Python.....	40
Figura 17 - Exemplo da utilização do comando <i>for</i> .....	40
Figura 18 - Sintaxe de funções em Python .....	41
Figura 19 - Sintaxe de classes em Python .....	42
Figura 20 - Código para criação de um mapa.....	53
Figura 21 - Exemplo de um mapa básico. ....	53
Figura 22 - Exemplo de eventos do Google Maps .....	55
Figura 23 - Oculta e exhibe controles no mapa .....	56
Figura 24 - Mapa sem os controles de navegação e com controle escala ativado.....	57
Figura 25 - Exemplo de sobreposição .....	58
Figura 26 - Exemplo de marcador e de janela de informações .....	59
Figura 27 - Exemplo de geocodificação a partir da entrada do usuário .....	60
Figura 28 - Exemplo de geocodificação .....	61
Figura 29 - Exemplo de rotas de dois pontos passados pelo usuário .....	62
Figura 30 - Exemplo de rota gerada .....	63
Figura 31 - Exemplo código para visualização do Street View .....	64
Figura 32 - Exemplo de mapa na visualização do Street View .....	65
Figura 33 - Arquitetura de uma aplicação geográfica com GeoDjango. ....	67
Figura 34 - Comando para descompactar e instalar o Django.....	68
Figura 35 - Comando para criar um novo projeto em Django.....	69
Figura 36 - Execução do servidor web do framework Django.....	70
Figura 37 - Acesso ao servidor de desenvolvimento do Django .....	70
Figura 38 - Comando para criar uma aplicação.....	71
Figura 39 - Configuração para adicionar uma aplicação ao projeto.....	72
Figura 40 - Configuração para utilização do banco de dados PostgreSQL.....	73
Figura 41 - Exemplo de definição de modelo.....	74
Figura 42 - SQL gerada a partir da definição do modelo .....	74
Figura 43 - Opções disponíveis no model do Django .....	75
Figura 44 - Sincronização das tabelas no banco de dados de acordo com o modelo de dados .....	76
Figura 45 - Exemplo de template pai .....	77



Figura 46 - Exemplo de template filho .....	77
Figura 47 - Exemplo de função view .....	78
Figura 48 - Configuração do arquivo urls.py .....	78
Figura 49 - Comando para iniciar o console interativo .....	79
Figura 50 - Definição do modelo de dados Aluno .....	79
Figura 51 - SQL gerada pelo Django para criação da tabela de Aluno .....	79
Figura 52 - Criação de um registro na tabela Aluno.....	80
Figura 53 - Atualização de um registro na tabela Aluno .....	81
Figura 54 - Consulta a todos os registros da tabela Aluno .....	82
Figura 55 - Consulta condicional aos registros da tabela Aluno .....	82
Figura 56 - Configuração para habilitar a interface de administração .....	84
Figura 57 - Comando para sincronizar criar as tabelas da área administrativa .....	84
Figura 58 - Configuração de URL para habilitar a interface de administração.....	85
Figura 59 - Tela de login da interface de administração do Django.....	85
Figura 60 - Tela principal da interface de administração do Django .....	86
Figura 61 - Arquivo admin.py .....	86
Figura 62 - Formulário para cadastro de aluno.....	87
Figura 63 - Arquivo admin.py com opções adicionais .....	88
Figura 64 - Listagem dos registros com os recursos habilitados no arquivo admin.py.....	89
Figura 65 - Formulário de cadastro de aluno com os campos ordenados pelo atributo fields .....	89
Figura 66 - Criação de uma base de dados utilizando o modelo do PostGIS.....	91
Figura 67 - Configuração do banco de dados da aplicação de demonstração .....	91
Figura 68 - Modelo de dados da aplicação de demonstração .....	92
Figura 69 - Arquivo admin.py da aplicação de demonstração .....	93
Figura 70 - Configuração do arquivo urls.py da aplicação de demonstração.....	93
Figura 71 - Página principal do sistema .....	94
Figura 72 - Página para inclusão de um ponto .....	95
Figura 73 - Ponto selecionado no mapa.....	96
Figura 74 - Visualização das informações de um ponto.....	96
Figura 75 - Tela de listagem dos registros.....	97
Figura 76 - Formulário de cadastro/edição.....	98

## LISTA DE ABREVIATURAS E SIGLAS

<b>API</b>	Application Programming Interface
<b>CSS</b>	Cascading Style Sheets
<b>DBA</b>	Data Base Administrator
<b>DRY</b>	Don't Repeat Yourself
<b>GIS</b>	Geographic Information System
<b>HTML</b>	HyperText Markup Language
<b>HTTP</b>	HyperText Transfer Protocol
<b>JSON</b>	JavaScript Object Notation
<b>KML</b>	Keyhole Markup Language
<b>MVC</b>	Model View Controller
<b>OGC</b>	OpenGIS Consortium
<b>ORM</b>	Object-Relational Mapping
<b>GEOJSON</b>	Geometry JavaScript Object Notation
<b>GML</b>	Geography Markup Language
<b>SFSS</b>	Simple Features Specification for SQL
<b>SGBD</b>	Sistema Gerenciador de Banco de Dados
<b>SIG</b>	Sistemas de Informação Geográfica
<b>SQL</b>	Structured Query Language
<b>SVG</b>	Scalable Vector Graphics
<b>URL</b>	Uniform Resource Locator
<b>W3C</b>	World Wide Web Consortium
<b>WFS</b>	Web Feature Services
<b>WKT</b>	Well-Known Text
<b>WKB</b>	Well-Known Binary

## SUMÁRIO

<b>1 INTRODUÇÃO .....</b>	<b>12</b>
<b>2 FUNDAMENTAÇÃO TEÓRICA.....</b>	<b>14</b>
2.1 APLICAÇÕES WEB.....	14
2.2 SISTEMAS DE INFORMAÇÃO GEOGRÁFICA .....	15
2.3 O PADRÃO MVC .....	17
2.4 DESENVOLVIMENTO WEB .....	18
<b>2.4.1 HTML .....</b>	<b>18</b>
<b>2.4.2 CSS.....</b>	<b>19</b>
<b>2.4.3 JAVASCRIPT.....</b>	<b>19</b>
<b>2.4.4 BANCO DE DADOS.....</b>	<b>20</b>
<b>2.4.5 FRAMEWORK.....</b>	<b>24</b>
2.5 LINGUAGEM PYTHON.....	25
<b>2.5.1 CARACTERÍSTICAS DA LINGUAGEM.....</b>	<b>26</b>
<b>2.5.2 VANTAGENS DA LINGUAGEM PYTHON .....</b>	<b>26</b>
<b>2.5.3 SINTAXE BÁSICA DA LINGUAGEM PYTHON.....</b>	<b>27</b>
2.6 DJANGO .....	42
<b>2.6.1 MTV .....</b>	<b>45</b>
<b>2.6.2 ORM.....</b>	<b>48</b>
<b>2.6.3 INTERFACE DE ADMINISTRAÇÃO.....</b>	<b>49</b>
2.7 GEODJANGO .....	49
2.8 GOOGLE MAPS .....	51
<b>2.8.1 API GOOGLE MAPS.....</b>	<b>51</b>
<b>3 DESENVOLVIMENTO.....</b>	<b>66</b>
3.1 DEFINIÇÃO DAS TECNOLOGIAS .....	66
3.2 PREPARAÇÃO DO AMBIENTE .....	67
3.3 CONFIGURAÇÃO DA ARQUITETURA PROPOSTA .....	68

<b>3.3.1 INSTALAÇÃO DA FERRAMENTA.....</b>	<b>68</b>
<b>3.3.2 INICIANDO UM PROJETO.....</b>	<b>69</b>
<b>3.3.3 CONFIGURANDO O PROJETO .....</b>	<b>72</b>
<b>3.3.4 MTV.....</b>	<b>74</b>
<b>3.3.5 ORM.....</b>	<b>79</b>
<b>3.3.6 INTERFACE DE ADMINISTRAÇÃO.....</b>	<b>84</b>
<b>3.4 DESENVOLVIMENTO DA APLICAÇÃO DE DEMONSTRAÇÃO .....</b>	<b>90</b>
<b>4 RESULTADOS OBTIDOS.....</b>	<b>94</b>
<b>5 CONSIDERAÇÕES FINAIS .....</b>	<b>99</b>
<b>REFERÊNCIAS .....</b>	<b>100</b>

# 1 INTRODUÇÃO

A utilização de aplicações de *softwares* baseadas na *web* está se tornando cada vez mais comum. Segundo Darie et al. (2006), esse crescimento é devido às várias vantagens que um sistema *web* pode oferecer, tais como: mobilidade, investimento reduzido, facilidade de manutenção, segurança, entre outros. Outros fatores que podem ser considerados são o avanço da Internet e o desenvolvimento das linguagens de programação para *web*.

Com a demanda crescente de projetos envolvendo novos produtos e processos baseados na *web*, em geral acompanhados de fortes restrições de prazos de entrega, os desenvolvedores perceberam a necessidade de novas ferramentas e metodologias que os apoiassem no desenvolvimento de aplicações em períodos curtos, porém mantendo os requisitos de qualidade e segurança das aplicações.

Com o intuito de resolver esses problemas, foram desenvolvidas soluções que viabilizassem diversas melhorias ao processo de codificação dos sistemas, essas soluções foram denominadas *frameworks* de desenvolvimento. Segundo Fayad, Schmidt e Johnson (1999) as principais melhorias consistem de: modularidade, reusabilidade, extensibilidade e inversão de controle.

O Django é um *framework web* de alto nível<sup>1</sup> escrito em Python<sup>2</sup> que estimula o desenvolvimento ágil.

Desenvolvimento ágil é um conjunto de metodologias de desenvolvimento de *software*. A metodologia ágil tenta minimizar o risco do desenvolvimento de *software* fazendo entregas em curtos períodos, chamando esses de períodos de iteração. O foco do desenvolvimento ágil é a satisfação do cliente, fazendo entregas rápidas e funcionais dos *softwares*. Segundo Fowler (2012) o desenvolvimento ágil se distingue do desenvolvimento tradicional na menor geração de documentos e criação de mais código, destacando que essa não é a principal característica do desenvolvimento ágil.

Assim como outros *frameworks* ágeis para desenvolvimento *web*, o Django utiliza o conceito de *Don't Repeat Yourself* (DRY) (Não se repita), que funciona com a ideia de usar convenções em substituição às configurações. Isso quer dizer que, se o desenvolver seguir

---

<sup>1</sup> Framework com nível de abstração relativamente elevado. O programador não precisa conhecer o processo que o framework utiliza para acessar o banco de dados, tratar uma requisição, salvar um dado. Essas características são abstraídas pelo framework de alto nível.

<sup>2</sup> Python é uma linguagem de programação interativa, interpretada e orientada a objetos (BUDD, 2009).

determinadas convenções na maneira de estruturar seu código, ele não precisará configurar características específicas do *software*.

O presente trabalho teve como objetivo geral demonstrar a viabilidade do desenvolvimento de aplicações geográficas, utilizando o *framework* Django/GeoDjango, juntamente com a *Application Programming Interface* (API) de mapas do Google Maps e a extensão espacial para o banco de dados PostgreSQL, o PostGIS, tendo ainda como objetivos específicos: apresentar a linguagem de programação Python, estudar o *framework* Django e suas convenções, bem como, sua extensão *Geographic Information System* (GIS), estudar a API do Google Maps, seus recursos e serviços disponíveis, estudar a extensão espacial PostGIS, reunir em um único documento as orientações necessárias e suficientes para se desenvolver uma aplicação e o desenvolvimento de uma aplicação de demonstração que agrupa todos os elementos citados.

Para se alcançar os objetivos do trabalho, a seguinte metodologia foi utilizada: revisão da bibliografia referente à linguagem de programação, ferramentas e bibliotecas utilizadas; desenvolvimento de uma aplicação de demonstração; análise da aplicação desenvolvida, verificando a viabilidade de desenvolver aplicações geográficas utilizando as tecnologias propostas.

A organização do presente trabalho é definida da seguinte forma: no capítulo 2 é feita a fundamentação teórica, explicando os assuntos referentes ao trabalho. No capítulo 3 são definidas as tecnologias, ferramentas e o ambiente necessário para o desenvolvimento de uma aplicação geográfica, bem como, as configurações e explicações relevantes sobre o *framework* Django e o desenvolvimento propriamente dito da aplicação geográfica de demonstração, mostrando a integração entre as tecnologias. No capítulo 4 são apresentados os resultados obtidos, seguido do capítulo 5 com as considerações finais.

## 2 FUNDAMENTAÇÃO TEÓRICA

Desenvolver um sistema de informações é uma tarefa complexa. Quando se trata de desenvolver de um sistema *web*, com foco na manipulação de dados geográficos, faz-se necessário reunir um conjunto significativo de conhecimentos envolvendo, desde padrões de projetos, até técnicas e linguagens de programação para *web*, bancos de dados, *frameworks* para desenvolvimento *web* e suas extensões etc. Assim, este capítulo se propõe a reunir os principais conceitos, técnicas e recursos envolvidos num projeto de desenvolvimento de uma aplicação geográfica *web* utilizando o *framework* Django. O entendimento sobre a linguagem Python é essencial para a compreensão das configurações e dos códigos do Django, tornando a leitura do capítulo 2.5 um requisito parcial para os capítulos subsequentes a respeito do *framework* Django.

### 2.1 APLICAÇÕES WEB

Segundo Deitel e Deitel (2008), no começo da internet, os conteúdos disponibilizados na rede mundial de computadores eram estáticos, com isso, a atualização dos conteúdos era feita somente de forma manual. A necessidade de atualizar os documentos publicados de maneira simples e rápida, fez com que surgissem tecnologias para disponibilização de conteúdos dinâmicos. Essa mudança fez com que os simples documentos de conteúdos fossem se transformando em verdadeiras aplicações, que armazenavam seus dados em um banco de dados e que permitiam a sua manipulação após serem disponibilizados na *web*.

Com essa mudança de paradigma a *web* foi se transformando, os usuários que eram somente espectadores começaram a fazer parte da *web* e interagir com ela, com isso foi criado o termo *Web 2.0*, onde o foco era a participação e colaboração dos usuários nos conteúdos dos *sites* e aplicações (DEITEL; DEITEL, 2008).

Segundo Gonçalves (2007), a *web* é uma aplicação cliente/servidor em grande escala, onde um cliente se conecta a um servidor utilizando um protocolo. O protocolo mais comum para essa comunicação é o *HyperText Transfer Protocol* (HTTP), usado em conjunto

com a linguagem *Hypertext Markup Language* (HTML) para exibição das páginas no navegador.

As aplicações *web* funcionam da seguinte maneira, o usuário envia uma requisição ao servidor e espera uma resposta. A requisição é processada pelo servidor e este devolve a resposta ao usuário (TANENBAUM; STEEN, 2007), conforme a Figura 1.

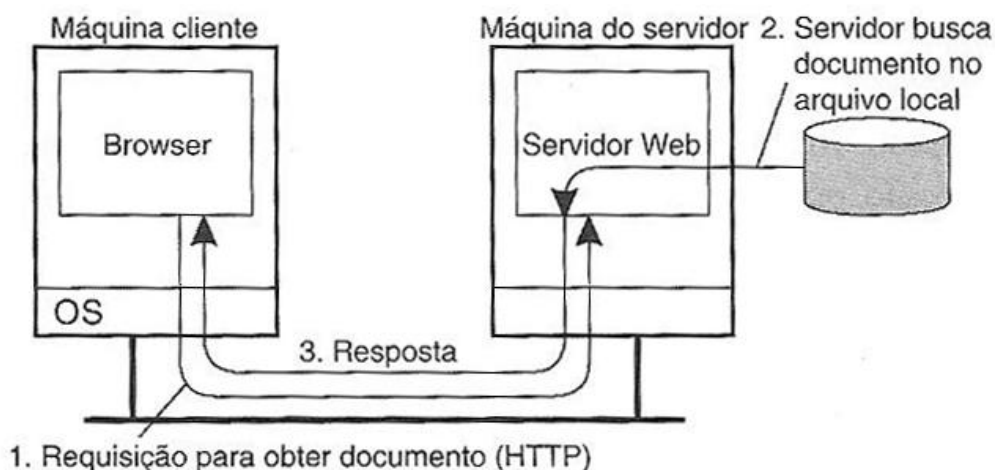


Figura 1 - Funcionamento padrão de uma aplicação web

Fonte: TANENBAUM e STEEN (2007, p.331)

## 2.2 SISTEMAS DE INFORMAÇÃO GEOGRÁFICA

Segundo Pick (2005), Sistemas de Informação Geográfica (SIG) são sistemas que acessam informações e atributos espaciais, as analisa e produzem saídas como a produção de mapas e exibição visual. O termo SIG é aplicado para sistemas que realizam a manipulação computacional de dados geográficos. Segundo Câmara et al. (2005) a principal diferença de um SIG para um sistema de informação convencional é sua capacidade de armazenar tanto os dados descritivos como as geometrias dos diferentes tipos de dados geográficos.

Segundo Câmara et al. (2005) as principais características de SIGs são:

- Inserir, integrar, armazenar, manipular, analisar e apresentar dados referidos espacialmente na superfície terrestre;
- Oferecer mecanismos para combinar, manipular, analisar, consultar, recuperar e visualizar o conteúdo da base de dados geográficos.



Segundo Câmara et al. (1996), SIGs comportam diferentes tipos de dados e aplicações, em várias áreas do conhecimento. Alguns exemplos são: otimização de tráfego, controle cadastral, gerenciamento de serviços de utilidade pública, demografia, cartografia, administração de recursos naturais, monitoramento costeiro, controle de epidemias e planejamento urbano.

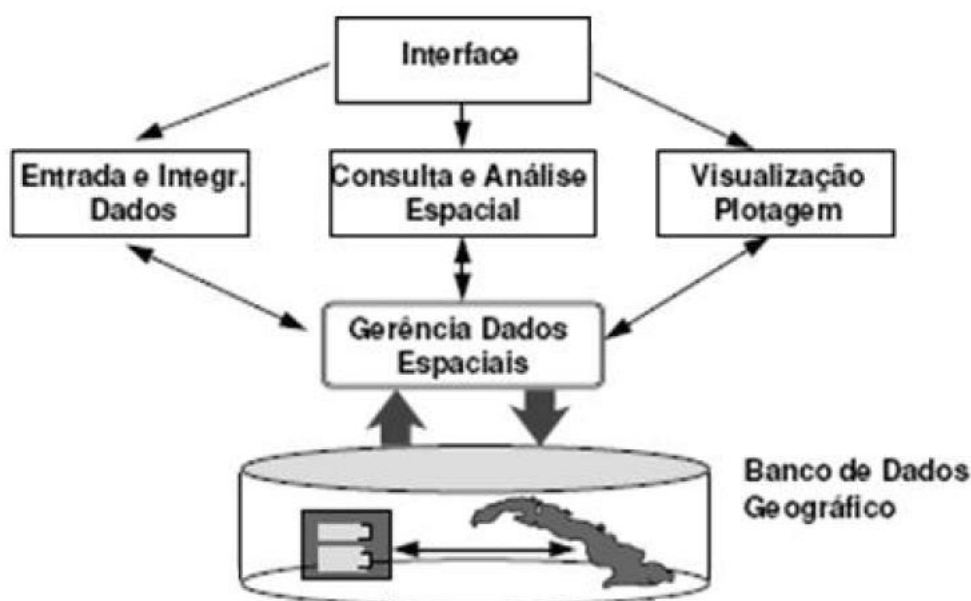


Figura 2 - Arquitetura de sistemas de informação geográfica

Fonte: CÂMARA et al (2005, p.13)

A Figura 2 apresenta os componentes de um SIG. Segundo Câmara et al.(2005), essa arquitetura de componentes é dividida em três níveis. O primeiro, mais próximo do usuário, é a interface. Ela que define como o sistema é operado e controlado. No nível intermediário, são apresentados mecanismos de processamento de dados espaciais, tais como: entrada de dados, que inclui os mecanismos de conversão de dados, algoritmos de consulta e análise espacial, visualização e plotagem. No nível mais interno do sistema, um sistema de gerência de banco de dados geográficos oferece armazenamento e recuperação de dados espaciais e seus atributos.

## 2.3 O PADRÃO MVC

O Modelo, Visão e Controlador (MVC) é um padrão de projeto utilizado no desenvolvimento de *software*.

Padrão de projeto é uma solução documentada de um problema, que pode ser reutilizada em várias outras situações. Cada padrão descreve quais as situações em que ele pode ser aplicado, quais as restrições e quais as consequências, custos e benefícios advindas da sua utilização (GAMMA, 2000).

A abordagem MVC é uma arquitetura utilizada para desenvolver interfaces com o usuário, que é composta de três partes. O *Model* (modelo) gerencia o comportamento dos dados da aplicação, a *View* (visão) gerencia a saída gráfica e textual da parte da aplicação que é visível ao usuário e o *Controller* (controlador) interpreta as entradas de mouse e teclado do usuário, ele comanda a visão e o modelo para se alterarem de forma apropriada (GAMMA, 2000).

Todas as requisições feitas pelo usuário são enviadas ao *Controller*, este manipula os dados usando o *Model* e invoca a *View* correta, de acordo com a ação executada.

A grande vantagem de se utilizar o padrão MVC é a separação de lógica e apresentação dos dados, sendo que isso favorece o trabalho em equipe. Um *designer* poderia trabalhar na apresentação, definindo o HTML, *Cascading Style Sheets* (CSS), Flash, enquanto um *Data Base Administrator* (DBA) poderia trabalhar com o modelo e outro programador poderia se concentrar nas regras de negócio inseridas no controlador. Dessa forma, qualquer mudança, por exemplo, na apresentação, teria pouco ou nenhum impacto nas demais camadas da aplicação.

Com essa abordagem é possível criar aplicações que funcionem em várias plataformas diferentes mudando apenas a lógica da camada de apresentação, um exemplo seria utilizar um sistema web por um computador pessoal e a partir de um celular, eles teriam camadas de apresentação diferentes, porém utilizariam a mesma camada de aplicação. A Figura 3 ilustra a interação entre as camadas da arquitetura MVC.

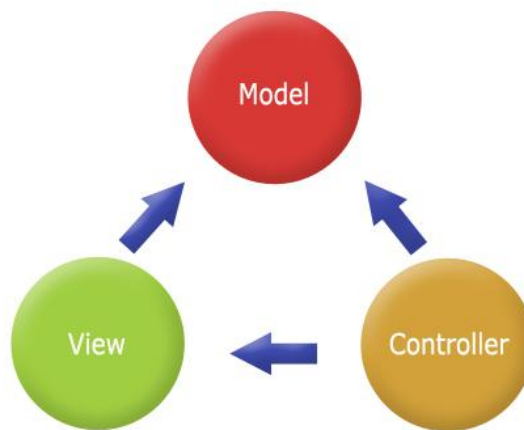


Figura 3 - Ilustração da interação entre as camadas da arquitetura MVC.  
Fonte: PRÓPRIA (2012)

## 2.4 DESENVOLVIMENTO WEB

Nesta seção serão apresentados os conceitos básicos para a construção de uma aplicação *web*. Conceitos que vão desde a estruturação e estilização de uma página *web* até a manipulação de um banco de dados.

### 2.4.1 HTML

A HTML é uma linguagem de marcação utilizada para criação de páginas *web*. Com ela é possível descrever a estrutura e formato do conteúdo de um documento *web*. A sigla HTML é um acrônimo para *Hypertext Markup Language*, que significa Linguagem de Marcação de Hipertexto(texto em formato digital) . A estrutura de um documento HTML é formada pelas etiquetas (*tags*) que definem a forma como o conteúdo é apresentado ao usuário.

O formato das etiquetas segue o seguinte padrão, primeiro é inserido o sinal de menor “<”, em seguida, o comando, e logo após, o sinal de maior “>”. As etiquetas limitam o escopo do seu comando quando é adicionado uma *tag* de finalização, que corresponde a *tag* inicial com a utilização de uma barra “/” antes do sinal de maior “>”.

As etiquetas essenciais para a elaboração de um documento *web* são:

- A *tag* `<HTML></HTML>`: todo documento deve possuir essa *tag*, ela que limita onde inicia e onde finaliza a estrutura do documento *web*;
- A *tag* `<HEAD></HEAD>`: essa *tag* serve para adicionar informações (*meta-data*) aos documentos, nela são especificados os elementos *scripts*, arquivos de estilos e o título da página;
- A *tag* `<BODY></BODY>`: todo o conteúdo da página fica dentro dessa *tag*. Essa é a parte do documento *web* que ficará disponível para o usuário acessar.

## 2.4.2 CSS

CSS é a linguagem utilizada para definir a apresentação dos documentos *web*. É ela que define a forma, cores, fontes, tamanho e posição dos elementos na página *web*. A sigla CSS é um acrônimo para *Cascading Style Sheets*, que significa Folhas de Estilo em Cascata.

A linguagem de marcação HTML é utilizada para determinar a estrutura do documento enquanto o CSS é utilizada para estilizar a aparência dessa estrutura.

A definição do CSS pode ficar em um arquivo independente ou pode ficar embutido na página HTML. A vantagem da utilização do arquivo independente é a possibilidade de utilizar a definição de estilo para várias páginas, enquanto a utilização do CSS embutido juntamente com o HTML pode ser usado apenas para um arquivo.

## 2.4.3 JAVASCRIPT

JavaScript é uma linguagem de *script* interpretada no navegador no momento da execução da página *web*. As linguagens de *script* são linguagens de programação que são executadas no interior de programas ou de outras linguagens de programação. Baseada na linguagem de programação ECMAScript, ela faz parte da construção da página *web* moderna. O HTML fornece a estrutura do documento *web*, o CSS adiciona os estilos as páginas e o JavaScript introduz as ações. Além de possibilitar a mudança de um estilo em tempo de

execução, ele monitora os eventos que acontecem na página, valida formulários, faz a interação com o usuário e ainda possibilita a comunicação assíncrona com o servidor.

Os arquivos JavaScript podem ser embutidos nas páginas HTML ou podem ficar em arquivos independentes. O uso de JavaScript em páginas HTML, pelo padrão *World Wide Web Consortium* (W3C), pode ser informado de duas maneiras ao navegador conforme mostrado na Figura 4.

```
<script type="text/javascript">  
    /* script */  
</script>  
<!-- ou -->  
<script type="text/javascript" src="arquivo.js ">  
</script>
```

Figura 4 - Exemplo de declaração de Javascript

Fonte: PRÓPRIA (2012)

#### **2.4.4 BANCO DE DADOS**

Um banco de dados é uma coleção de dados relacionados. Os dados são fatos que podem ser gravados e que possuem um significado implícito. Um banco de dados pode ser gerado e mantido manualmente ou pode ser computadorizado (ELMASRI; NAVATHE, 2011). Os bancos de dados computadorizados são mantidos por ferramentas conhecidas como Sistema Gerenciador de Banco de Dados (SGBD), que são um conjunto de ferramentas que permitem aos usuários criarem e manterem um banco de dados. O propósito do SGBD é facilitar a criação, manutenção, definição e compartilhamento de bancos de dados entre vários usuários e aplicações.

##### **2.4.4.1 POSTGIS**

O PostGIS é uma extensão espacial gratuita e de código fonte livre desenvolvida pela Refrations Research Inc, que permite o uso de objetos GIS no banco de dados

PostgreSQL (POSTGIS, 2012). O PostGIS adiciona vários tipos de dados espaciais e mais de 300 funções para trabalhar com esses tipos espaciais (OBE; HSU, 2011)

Os objetos de GIS suportados pelo PostGIS são um superconjunto do "Simple Features", definida pelo *OpenGIS Consortium* (OGC).

O *Open Geospatial Consortium* é um consórcio com mais de 450 companhias, agências governamentais e universidades, criado para promover o desenvolvimento de tecnologias que facilitem a interoperabilidade entre sistemas envolvendo informação espacial e localização (OGC, 2012).

A partir da versão 0.9 o PostGIS suporta todos os objetos e funções especificadas no padrão OGC chamado de *Simple Features Specification for SQL* (SFSS). A especificação OpenGIS da OGC define dois padrões de expressão de objetos espaciais: o *Well-Known Text* (WKT) e o *Well-Known Binary* (WKB). Ambos incluem informações sobre o tipo do objeto e as coordenadas que formam o objeto (POSTGIS, 2012).

O PostGIS oferece suporte aos seguintes tipos de objetos espaciais especificados na representação de texto (WKT):

- POINT(0 0)
- LINESTRING(0 0,1 1,1 2)
- POLYGON((0 0,4 0,4 4,0 4,0 0),(1 1, 2 1, 2 2, 1 2,1 1))
- MULTIPOINT(0 0,1 2)
- MULTILINESTRING((0 0,1 1,1 2),(2 3,3 2,5 4))
- MULTIPOLYGON((((0 0,4 0,4 4,0 4,0 0),(1 1,2 1,2 2,1 2,1 1)), ((-1 -1,-1 -2,-2 -2,-2 -1,-1 -1)))
- GEOMETRYCOLLECTION(POINT(2 3),LINESTRING(2 3,3 4))

A principal razão para o PostgreSQL ter sido escolhido como a plataforma sobre a qual construir o PostGIS foi a facilidade de extensibilidade que forneceu e a possibilidade de construir novos tipos e operadores, além do controle sobre as operações de indexação.

O PostgreSQL possui a reputação de ser um banco de dados de código aberto avançado. Ele possui a velocidade e as funcionalidades necessárias para competir com as principais ofertas de empresas comerciais disponíveis no mercado (OBE; HSU, 2011)

A seguir estão alguns recursos interessantes que ele possui e que não estão presentes nos demais bancos de dados abertos e em muitos bancos de dados comerciais também (OBE, 2011):

- Possibilidade de escolher várias linguagens de programação para escrever funções de banco de dados. Entre elas estão: *Structured Query Language* (SQL), PGSQL, C, Perl, Python, Java. Além disso, a plataforma PostgreSQL é mais extensível de qualquer plataforma de banco de dados, tornando mais fácil registrar *drivers* de novas linguagens;
- Assim como no Oracle e no IBM DB2 o PostgreSQL permite que o desenvolvedor defina qualquer coluna da tabela para que seja composta de *arrays* de *strings*, números, datas, geometrias ou tipos definidos por ele;
- O PostgreSQL tem um recurso chamado herança de tabela, similar aos conceitos da orientação a objetos, onde uma tabela pode herdar a estrutura de uma tabela pai. Além da herança simples o PostgreSQL suporta herança múltipla, onde uma tabela filho pode ter mais de uma tabela pai com colunas derivadas de ambos os pais.

#### 2.4.4.1.1 TIPOS DE DADOS

O PostGIS estende o PostgreSQL adicionando um tipo de dado chamado *geometry*. A maior parte das funções do PostGIS trabalha com os tipos geométricos básicos, tais como, pontos, linhas, polígonos e suas variações. O tipo de dado *geometry* pode ser tratado como qualquer outro tipo de dado nativo do PostgreSQL, tal como datas, números e texto (OBE; HSU, 2011)

Segundo Obe e Hsu (2011), o tipo *geometry* segue a definição de geometria do padrão OGC. Isso permite que todo o conhecimento adquirido no trabalho com um banco de dados espacial pode ser aplicado sobre qualquer outro banco de dados que siga esse padrão.

O PostgreSQL possui nativamente o tipo de dado *geometry*. Este tipo é incompatível com o tipo de dado *geometry* do PostGIS, além de ter pouco ou nenhum suporte a ferramentas de terceiros. Além disso, ele não segue os padrões OGC e também não apoia os sistemas de coordenadas espaciais.

#### 2.4.4.1.2 FUNÇÕES DE SAÍDA

Segundo Obe e Hsu (2011) as funções de saída são funções que retornam uma representação de geometria em outro formato padrão da indústria de aplicações de informação espacial e de localização. Essas funções geram saídas no formato de texto ou de arquivos, possibilitando a recuperação dos dados do banco de dados espacial em um outro formato. Entre as principais representações, estão:

- *Well-Known Text e Well-Known Binary*: *Well-known Text* é tipo mais comum dos formatos padronizados do OGC para geometrias. A função que gera essa saída é a ST\_AsEWKT. A função ST\_AsEWKT é uma extensão específica do PostGIS e não atende as definições do padrão OGC. *Well-Known Binary* é outro formato padrão do OGC. A função que gera a saída desse formato é a ST\_AsEWKB. Da mesma maneira que a função ST\_AsEWKT a função ST\_AsEWKB é uma extensão específica do PostGIS e não atende as definições do padrão OGC. Diferente do *Well-Know Text*, o *Well-Knowm Binary* mantém precisão, isso significa que, o dado que está armazenado no banco de dados é o mesmo da saída. As funções que atendam as especificações da OGC para o formato WKT e WKB, são respectivamente, a ST\_AsText e a ST\_AsBinary.
- *Keyhole Markup Language (KML)*: é um formato baseado em XML criado pela empresa Keyhole Corporation para renderizar suas aplicações. O KML ganhou popularidade depois que o Google adquiriu a empresa Keyhole Corporation e integrou o KML aos seus produtos Google Maps e o Google Earth. Recentemente o OGC aceitou o KML como um formato de transporte padrão. A função que gera essa saída é a ST\_AsKML;
- *Geography Markup Language (GML)*: é um formato baseado em XML definido pela OGC para transporte de dados. É comumente usado em *Web Feature Services (WFS)*, que é um serviço especificado pela OGC para acesso e manipulação de dados geográficos na web. A função que gera essa saída é a ST\_AsGML;
- *Geometry JavaScript Object Notation (GEOJSON)*: é um formato recentemente desenvolvido com base no *JavaScript Object Notation (JSON)*.



GeoJSON é voltado para a utilização de aplicações orientadas por Ajax. O JSON é a representação padrão de estruturas de dados em JavaScript. A função que gera essa saída é a `ST_AsGeoJSON`;

- *Scalable Vector Graphics* (SVG): é um formato popular para representação de imagens, suportado nas principais ferramentas de renderização e desenho. Pode ser convertido através de ferramentas para outros formatos de imagens. A maioria dos navegadores suporta, seja nativamente ou via *plug-in* instalável. A função que gera essa saída é a `ST_AsSVG`;
- *Geohash*: é um sistema de geocodificação utilizado para troca de coordenadas e não é aconselhado para representação visual dos dados espaciais.

No QUADRO 1 é mostrada a saída de todas as funções apresentadas nessa seção:

QUADRO 1  
Exemplos das saídas das funções

<i>Formato</i>	<i>Saída</i>
GML	<code>&lt;gml:LineString srsName="EPSG:4326"&gt; &lt;gml:coordinates&gt;-2,48 1,51&lt;/gml:coordinates&gt;&lt;/gml:LineString&gt;</code>
KML	<code>&lt;LineString&gt;&lt;coordinates&gt;2,48 1,51&lt;/coordinates&gt;&lt;/LineString&gt;</code>
GeoJSON	<code>{"type": "LineString", "coordinates": [[2,48], [1,51]]}</code>
SVG (Absolute)	<code>M 2 -48 L 1 -51</code>
SVG (Relative)	<code>M 2 4 L -1 -3</code>
Geohash	<code>u</code>

Fonte: OBE (2011, p.87)

Informações adicionais a respeito dos parâmetros de entrada e das saídas das funções dessa seção podem ser encontradas na documentação oficial do PostGIS disponível no endereço <http://postgis.refrations.net/documentation/>.

### 2.4.5 FRAMEWORK

Para desenvolver aplicações *web* os desenvolvedores utilizam diversos recursos para poupar o tempo de desenvolvimento. Com esse intuito, surgiram os *frameworks*. Os

*frameworks* de desenvolvimento de aplicações são uma estrutura de suporte bem definida em que outros projetos de *software* podem ser organizados e desenvolvidos. Esse tipo de *framework* possui a característica de diminuir o tempo de desenvolvimento associado às atividades comuns relacionadas ao desenvolvimento (SAUVÉ, 2012). Por exemplo, vários *frameworks* possuem ferramentas para acesso ao banco de dados, controle de sessão e autenticação de usuários.

O *framework* tenta fazer generalizações sobre as tarefas comuns e sobre o fluxo de trabalho de um problema específico. Com isso, o *framework* propõe uma plataforma sobre o qual as aplicações desse domínio podem ser construídas mais rapidamente (BROWN et al., 2008).

Segundo Sauv  (2012) os frameworks possuem as seguintes caracter sticas:

- Ser reutiliz vel;
- Bem documentado;
- F cil de usar;
- Deve ser extens vel;
- Ser seguro;
- Deve ser eficiente;
- Deve ser completo.

## 2.5 LINGUAGEM PYTHON

Segundo Catunda (2001), Python   uma linguagem de programac o simples e poderosa. Possui mecanismos eficientes e com um bom n vel de abstrac o para manipulac o de estrutura de dados.

O Python   uma linguagem interativa, interpretada, de alto n vel, orientada a objetos, multi-paradigma, com uma sintaxe elegante e tipagem din mica (BUDD, 2009).

Foi desenvolvida por Guido van Rossum em meados da d cada de 1990, nos laborat rios da CWI (Instituto Nacional de Pesquisa em Matem tica e Ci ncia da Computac o), localizado em Amsterd  (DEITEL et al., 2002).

O interpretador Python e suas bibliotecas padrões estão disponíveis, tanto o código-fonte como o binário no *site* <http://www.python.org> e podem ser livremente distribuída.

### **2.5.1 CARACTERÍSTICAS DA LINGUAGEM**

A característica de tipagem dinâmica permite que o tipo das variáveis seja especificado no momento da execução, o desenvolvedor não é solicitado a definir o tipo da variável antes de utilizá-la, a linguagem compreende qual o tipo do valor que está sendo atribuído à variável e determina o seu tipo, dinamicamente. Outra característica é a tipagem forte, ou seja, após especificar o tipo da variável não é possível fazer operações que não estejam relacionadas a esse tipo, um exemplo seria tentar concatenar dois números inteiros, gerando uma exceção, já que a operação de concatenação está relacionada ao tipo *string*.

### **2.5.2 VANTAGENS DA LINGUAGEM PYTHON**

Segundo Deitel et al. (2002, p. 15, tradução: PRÓPRIA)

“A linguagem Python oferece uma série de vantagens. É uma linguagem excelente para iniciantes em programação e para o desenvolvimento de aplicações comerciais, foi desenvolvida com o objetivo de ser portátil e extensível, a sua sintaxe promove o uso de boas práticas de programação, diminui o tempo utilizado no desenvolvimento de aplicações sem ignorar questões importantes para a manutenção das mesmas.”

Segundo Lutz e Ascher (2007, p. 36)

“ O Python possui as seguintes vantagens técnicas:

- É orientado a objetos: O Python é completamente orientado a objetos, suportando noções avançadas, como polimorfismo, sobrecarga de operadores e herança múltipla;
- É gratuito: O Python é gratuito e de código-fonte aberto. Ele é disponível gratuitamente na Internet e não existe nenhuma restrição para cópia;

- É portátil: O Python é portátil, compila e executa em praticamente todas as principais plataformas em uso atualmente. Os programas em Python que usam a linguagem básica e as bibliotecas padrão são executados igualmente em sistemas Unix, MS Windows e na maioria dos outros, com um interpretador Python;
- É poderoso: Na perspectiva de recursos, o Python é um pouco híbrido. Seu conjunto de ferramentas o posiciona entre as linguagens de script tradicionais (como Tel, Scheme e Perl) e as linguagens de desenvolvimento de sistemas (como C, C++ e Java);
- Pode ser misturado: Os programas em Python podem ser facilmente "colados" em componentes escritos em outras linguagens, de diversas maneiras. Por exemplo, a API C do Python permite que programas em C chamem e sejam chamados por programas em Python, de forma flexível;
- É fácil de usar: Para executar um programa em Python é relativamente simples, após a codificação basta executar o arquivo, não há etapas de compilação e ligação intermediárias, como acontece em linguagens como C ou C++;
- É fácil de aprender: Comparada com outras linguagens de programação, a linguagem Python básica é muito fácil de aprender.”

### 2.5.3 SINTAXE BÁSICA DA LINGUAGEM PYTHON

A linguagem Python tem uma sintaxe simples, baseada em instruções. Segundo Lutz e Ascher (2007) e Santana e Galesi (2010) a linguagem Python segue o seguinte conjunto de regras de sintaxe.

- Limite de blocos e de instruções: Na linguagem Python não existem chaves ou delimitadores tipo "*begin/end*" em torno de blocos de código. Ao invés disso, o Python usa a indentação de instruções sob um cabeçalho para agrupar as instruções em um bloco aninhado;
- Instruções compostas: No Python, as instruções compostas seguem um padrão: uma linha de cabeçalho terminada com dois-pontos, seguida de uma ou mais instruções aninhadas, normalmente indentadas sob o cabeçalho. Na instrução *if* as cláusulas *elif* e *else* fazem parte dela, mas são linhas de cabeçalhos que possuem seus próprios blocos aninhados;
- Expressões ignoradas: Normalmente, linhas em branco, espaços e comentários são ignorados. Os espaços brancos dentro de instruções e expressões são quase sempre ignorados, exceto quando estão em *strings* e em indentação. Os comentários são sempre ignorados, eles são iniciados com um caractere "#" e são finalizados com o final da linha;
- *Docstrings*: O Python suporta uma forma de comentário adicional chamada de *string* de documentação (abreviada de *docstring*), que podem ser *strings* normais

ou de multilinha . As *strings* de documentação tem a função de documentar módulos, funções, classes e método em Python. Seu conteúdo é ignorado pelo Python, contudo, elas são anexadas automaticamente aos objetos em tempo de execução e podem ser acessadas com ferramentas de documentação. A seção 2.5.3.1.1.2 apresenta a definição de *strings*, bem como, a sua sintaxe na linguagem Python.

### 2.5.3.1 VARIÁVEIS DO PYTHON

Python é uma linguagem dinamicamente tipada. Isso quer dizer que o programador não é obrigado a definir o tipo da variável antes de utilizá-la. A própria linguagem interpreta o tipo do valor que se deseja associar à variável e determina o seu tipo dinamicamente. Outra característica do Python é que ele permite associar às variáveis, não só os tipos básicos (números, *strings*, *booleanos*) como qualquer outro valor (funções, módulos, classes). Para saber qual o tipo de determinado objeto, usamos a função `type (var)`, que retorna o tipo do objeto *var*. Conforme a Figura 5.

A utilização do símbolo `>>>` nos exemplos dessa seção é para indicar que os códigos estão sendo executados no *shell* interativo da linguagem Python. A versão do Python utilizada nos exemplos é a 2.7.

```
1  >>> var = 1
2  >>> var
3  1
4  >>> type(var)
5  <type 'int'>
6  >>> var = 'texto'
7  >>> var
8  'texto'
9  >>> type(var)
10 <type 'str'>
```

Figura 5 - Exemplos de tipagem dinâmica

Fonte: PRÓPRIA (2012)

### 2.5.3.1.1 TIPOS BÁSICOS

Os principais tipos de objetos em Python são números, *strings* (texto), listas, tuplas e dicionários. Os conceitos tratados nessa seção são fortemente baseados nos conceitos apresentados por Santana e Galesi (2010).

#### 2.5.3.1.1.1 NÚMEROS

Os objetos numéricos do Python podem ser usados de maneira literal ou como retorno de alguma função ou operação aritmética.

A linguagem Python distingue objetos numéricos inteiros, de ponto flutuante e números complexos. Além disso, os valores booleanos também podem ser representados como números.

##### a) Inteiros (int)

Números inteiros podem ser representados em decimal, octal ou em hexadecimal. Além dos números inteiros convencionais, existem também os inteiros longos, que tem dimensão arbitrária e são limitados pela memória disponível para o interpretador. Quando um número inteiro normal excede a faixa dos números inteiros (int), a linguagem Python converte automaticamente esse inteiro para um inteiro longo. Na Figura 6 são mostrados alguns exemplos de números inteiros no Python.

```
1 >>> type(45)
2 <type 'int'>
3 >>> type(0xAF) # Hexadecimal
4 <type 'int'>
5 >>> type(056) # Octal
6 <type 'int'>
7 >>> 45, 0xAF, 056
8 (45, 175, 46)
```

Figura 6 - Exemplo de números inteiros em Python

Fonte: PRÓPRIA (2012)

**b) Ponto flutuante (float)**

Os números de ponto flutuante são capazes de representar qualquer número real independentemente do tamanho, limitando apenas pela quantidade de memória disponível para o interpretador. Os números de ponto flutuante podem ser representados de duas formas: normal (1.7) e por meio de notação científica (7e-5). Na Figura 7 são mostrados alguns exemplos de números de ponto flutuante no Python.

```
1 >>> type(1.9)
2 <type 'float'>
3 >>> type(7e-5)
4 <type 'float'>
5 >>> type(1e10)
6 <type 'float'>
7 >>> 7e-4
8 0.0007
9 >>> 7e-5
10 7e-05
11 >>> 7e-5 - 7e-4
12 -0.00063
```

Figura 7 - Exemplos de números de ponto flutuante em Python

Fonte: PRÓPRIA (2012)

### c) Números complexos (complex)

Números complexos são aqueles que possuem uma parte real e uma parte imaginária. São representados por dois números de ponto flutuante que compõem a parte real e imaginária do número complexo. Na Figura 8 são mostrados alguns exemplos de números complexos no Python.

```
1 >>> type(1.5+3.4j)
2 <type 'complex'>
3 >>> type(1.7+3.5j + 2.3+6.5j)
4 <type 'complex'>
5 >>> print 5.1+4.5j
6 (5.1+4.5j)
7 >>> print(1.7+3.5j + 2.3+6.5j)
8 (4+10j)
```

Figura 8 - Exemplos de números complexos em Python

Fonte: PRÓPRIA (2012)

### d) Booleanos (bool)

Na linguagem Python, os valores booleanos são representados como números inteiros, ou seja, qualquer valor zero significa “Falso”, e qualquer outro valor diferente de zero significa “Verdadeiro”. Além disso, existem as constantes *True* e *False* que podem ser usadas para essa função.

O Python adota que determinados valores são falsos. Alguns desses valores são:

- *None*;
- Zero de qualquer tipo numérico: 0, 0L, 0.0, ...;
- *Strings* vazias: “” ou “”;
- Sequências vazias: () ou [];
- Dicionário vazio: {};
- Conjuntos vazios: set([]);
- Instância de classes de usuários cujos métodos `__nonzero__()` ou `__len__` retornem zero.



Qualquer outro valor diferente desses é considerado verdadeiro na linguagem Python.

Na Figura 9 são mostrados alguns exemplos de valores booleanos no Python.

```
1 >>> type(True)
2 <type 'bool'>
3 >>> type(False)
4 <type 'bool'>
5 >>> bool(0)
6 False
7 >>> bool(1)
8 True
9 >>> bool([])
10 False
11 >>> bool(5)
12 True
```

Figura 9 - Exemplos de valores booleanos

Fonte: PRÓPRIA (2012)

#### 2.5.3.1.1.2 **STRINGS**

Variáveis do tipo *string* armazenam cadeias de caracteres como nomes e textos em geral. Elas podem ser representadas por uma coleção de caracteres entre aspas duplas (") ou simples(') e por três aspas duplas("''") ou simples('''), a qual possibilita criar *strings* multilinhas.

As *strings* são imutáveis, ou seja, uma vez criadas, elas não podem ser modificadas. Portanto não é possível adicionar, remover ou mesmo modificar algum caractere de uma *string*. Dessa forma, por exemplo, para cada concatenação de uma *string* com outra, um novo objeto é criado na memória e os anteriores perderão suas referências. Para acessar os caracteres de uma *string* deve-se informar qual o índice ou posição do caractere entre colchetes ([índice]).

As variáveis do tipo *string* suportam algumas operações tais como: fatiamento, concatenação e composição. O fatiamento é a capacidade de usar apenas parte de uma *string*. A concatenação é a operação de juntar duas ou mais *strings* em uma nova *string*. A composição é a utilização de *strings* como modelos onde é possível inserir outras *strings*. A Figura 10 ilustra a utilização das variáveis de tipo *string* bem com as suas operações.

```
1 >>> type('texto')
2 <type 'str'>
3 >>> "Hello World"[1]
4 'e'
5 >>> var = "Hello World"
6 >>> #exemplos de fatiamento
7 >>> var[1], var [4], var[7]
8 ('e', 'o', 'o')
9 >>> var[1], var [4], var[8]
10 ('e', 'o', 'r')
11 >>> var[1:5]
12 'ello'
13 >>> var[0:5]
14 'Hello'
15 >>> var[6:]
16 'World'
17 >>> var[:5]
18 'Hello'
19 >>> #exemplos de concatenação
20 >>> var = 'texto'
21 >>> var1 = 'abcd'
22 >>> var + var1
23 'textoabcd'
24 >>> var + "A" * 4
25 'textoAAAA'
26 >>> var + var*4
27 'textotextotextotextotexto'
28 >>> #exemplos de composição
29 >>> "João comprou %d laranjas" % 20
30 'João comprou 20 laranjas'
31 >>> "Maria comprou %d laranjas" % 10
32 'Maria comprou 10 laranjas'
33 >>> "%s tinha R$%5.2f e comprou %d frutas" % ("Maria",12.50,10)
34 'Maria tinha R$12.50 e comprou 10 frutas'
```

Figura 10 - Exemplos de strings em Python

Fonte: PRÓPRIA (2012)

### 2.5.3.1.1.3 LISTAS

Listas são coleções heterogêneas de objetos, que podem ser de qualquer tipo, inclusive de outras listas. As listas são representados entre colchetes “[ ]” e seus elementos são divididos por vírgula “,”. As listas são sequências mutáveis, ou seja, depois de criadas, elas podem alterar o valor atribuído a um determinado endereço da lista. As listas podem ser fatiadas da mesma forma que as *strings*, porém como são mutáveis, é possível fazer operações de atribuições, podendo alterar os valores individuais da lista. A Figura 11 ilustra a utilização das variáveis do tipo lista bem com as suas operações.

```
1 >>> type([])
2 <type 'list'>
3 >>> ['item1','item2','item3']
4 ['item1', 'item2', 'item3']
5 >>> ['item1','item2','item3'] + ['item4'] # concatenação
6 ['item1', 'item2', 'item3', 'item4']
7 >>> a = ['a1', 'a2', 'a3']
8 >>> a[0] #acessando item da lista
9 'a1'
10 >>> a[1] #acessando item da lista
11 'a2'
12 >>> a[0:2]
13 ['a1', 'a2']
14 >>> a[0:2] #fatiamento
15 ['a1', 'a2']
16 >>> lista ['string',1,1+3j]
17 >>> type(lista[0]),type(lista[1]),type(lista[2])
18 (<type 'str'>, <type 'int'>, <type 'complex'>)
```

Figura 11 - Exemplos de lista em Python

Fonte: PRÓPRIA (2012)

### 2.5.3.1.1.4 TUPLAS

As tuplas funcionam de maneira semelhante às listas, porém são sequências imutáveis, assim não podem acrescentar, remover ou fazer atribuições aos seus elementos e também não podem ter o seu tamanho alterado depois da criação. Da mesma maneira que a

lista as tuplas também podem armazenar quaisquer objetos; inteiros, *floats*, listas, outras tuplas, etc.

Um aspecto importante da tupla é que os objetos que estão dentro dela ainda mantêm as suas características. Então, se um elemento da tupla for uma lista, ela poderá ser modificada sem nenhum problema.

As tuplas são mais simples e eficientes do que as listas convencionais, portanto consomem menos recursos do computador.

“As tuplas são representadas entre parênteses “( )” e seus elementos são divididos por vírgula “,”, tal como nas listas. A Figura 12 ilustra a utilização das variáveis do tipo tupla bem com as suas operações.

```

1 >>> type(())
2 <type 'tuple'>
3 >>> a = ('item1','item2','item3','item4')
4 >>> a[0]
5 'item1'
6 >>> a[1]
7 'item2'
8 >>> a[1] = 'item modificado'   # tupla é imutável e não permite modificar seus elementos
9 Traceback (most recent call last):
10 File "<pyshell#6>", line 1, in <module>
11 TypeError: 'tuple' object does not support item assignment #mensagem de erro do
    interpretador quando um item da tupla sofre uma operação de atribuição
12 >>> tupla = ([1,2,3], 'elemento')
13 >>> tupla[0].append(4) # o elemento do índice 0 da tupla é um lista, sendo assim,
14 é possível adicionar, remover ou modificar seus elementos
15 >>> tupla
16 ([1, 2, 3, 4], 'elemento')
```

Figura 12 - Exemplo de tuplas em Python

Fonte: PRÓPRIA (2012)

#### 2.5.3.1.1.5 DICIONÁRIOS

Dicionários são estruturas que armazenam conjuntos de objetos de maneira semelhante a uma lista ou tupla. A diferença entre o dicionário de uma sequência é a forma como esses objetos são indexados, ou seja, enquanto as sequências são indexadas por inteiros,

dicionários são indexados por chaves, que podem ser de qualquer tipo imutável (como *strings*, inteiros e tuplas).

Os dicionários são representados entre chaves “{}” e seus elementos são divididos por vírgula “,”, e cada elemento do dicionário é formado por uma combinação de chave e valor, separados por um dois pontos “:”.

Os dicionários são estruturas mutáveis, ou seja, depois de criados, eles podem alterar o valor atribuído a um determinado elemento. Contudo, por não se tratar de uma sequência não é possível executar as operações de fatiamento.

Os dicionários são estruturas de dados mais complexas e ocupam mais espaço na memória se comparado às estruturas de sequências, por isso, seu uso é indicado somente em situações que eles realmente forem necessários.

O dicionário é acessado por suas chaves. Para obter o valor de um elemento, é necessário escrever o nome do dicionário seguido da sua chave, *variável[key]*, onde *variável* é do tipo dicionário e *key* é a chave. Diferente das sequências, onde o índice é um número, dicionários utilizam chaves como índice. Quando um valor é atribuído a uma chave, duas coisas podem ocorrer: Se a chave já existe no dicionário, o seu valor será alterado para o novo valor. Se a chave não existe no dicionário, a nova chave será adicionada ao dicionário.

A Figura 13 ilustra a utilização das variáveis do tipo dicionário bem com as suas operações.

```
1 >>> dic = {'key1': 'value1'}
2 >>> dic
3 {'key1': 'value1'}
4 >>> dic['key2'] = 'value2'
5 >>> dic
6 {'key2': 'value2', 'key1': 'value1'}
7 >>> tupla = (1,2)
8 >>> dic[tupla] = "tupla como chave"
9 >>> dic
10 {(1, 2): 'tupla como chave', 'key2': 'value2', 'key1': 'value1'}
11 >>> dic["key1"]
12 value1'
```

Figura 13 - Exemplos de dicionários em Python

Fonte: PRÓPRIA (2012)

### 2.5.3.2 PRECEDÊNCIA DE OPERADORES EM PYTHON

A precedência de um operador especifica quem tem mais prioridade em uma expressão que utilize mais de 2 operadores juntos. Por exemplo, na expressão,  $2 + 4 * 3$ , a resposta é *14* e não *18* porque o operador de multiplicação (" $*$ ") tem maior prioridade de precedência que o operador de adição (" $+$ "). Para tornar uma expressão mais legível são utilizados parênteses.

O QUADRO 2 resume as precedências dos operadores para a linguagem Python, do de menor precedência (*least binding*) ao de maior precedência (*most binding*).

QUADRO 2

Precedência de operadores

(Continua)

Operador	Descrição
lambda	Expressão Lambda
if – else	Expressão Condicional
or	Operador Booleano OR
and	Operador Booleano AND
not x	Operador Booleano NOT
in, not in, is, is not, <, <=, >, >=, <>, !=, ==	Comparações, testes de membros/existência e testes de identidade
	Operador bit-a-bit OR
^	Operador bit-a-bit XOR
&	Operador bit-a-bit AND
<<, >>	Deslocamentos de bits
+, -	Adição e subtração
*, /, //, %	Multiplicação, divisão, divisão inteira e resto
+X, -X, ~X	Positivo, negativo, operador bit-a-bit NOT
**	Exponenciação
x[index], x[index:index], x(arguments...), x.attribute	Subscrição, repartição, chamada de função, referencia de atributo
(expressions...), [expressions...], {key:datum...}, `expressions...`	União ou exibição de tupla, exibição de listas, exibição de dicionários, conversão de texto

Fonte: PYTHON (2012)

### 2.5.3.3 ESTRUTURA DE CONTROLE DE FLUXO

No desenvolvimento de um programa são executadas uma série de instruções que nem sempre são as mesmas ou não são executadas na mesma ordem. Alguns trechos do código somente são executados se uma certa condição for verdadeira, para esses casos, existem as estruturas de controle de fluxo, que servem para determinar a forma de execução de um comando ou de uma sequência de instruções, às vezes chamada de "bloco".

Como em outras linguagens de programação a linguagem Python também possui seus controles de fluxos bem como uma sintaxe bem definida para cada um. Nessa seção serão apresentados os comandos utilizados nas operações de iteração e decisão.

#### 2.5.3.3.1 OS COMANDOS IF, ELIF, ELSE

O comando *if/elif/else* é o comando de decisão da linguagem Python. É muito comum em um programa que certos conjuntos de instruções sejam executados de forma condicional.

A sintaxe é composta por um teste *if*, seguido de zero ou mais testes *elif* e termina com um bloco *else* opcional.

Na Figura 14 é mostrada a sintaxe para o comando *if* do Python.

```
if <condição>:  
    <bloco de código>  
elif <condição>:    #opcional  
    <bloco de código>  
elif <condição>:    #opcional  
    <bloco de código>  
else:                #opcional  
    <bloco de código>
```

Figura 14 - Sintaxe do comando *if* em Python

Fonte: PRÓPRIA (2012)

#### 2.5.3.3.2 O COMANDO WHILE

O comando *while* é responsável pela criação de iterações em Python. A sua sintaxe é composta por uma expressão e um bloco de códigos. O funcionamento dele consiste em executar o bloco de códigos enquanto uma expressão for verdadeira. Caso a interação seja executada até o fim sem ser interrompido por um comando *break*, os comandos do bloco de código do *else* serão executados, sendo este opcional.

O comando *while* é indicado quando não há como determinar o número de iterações que vão ocorrer.

Na Figura 15 é mostrada a sintaxe para o comando *while* do Python.

```
while <condição>:  
    <bloco de código>  
else:                #opcional  
    <bloco de código>
```

Figura 15 - Sintaxe do comando *while* em Python

Fonte: PRÓPRIA (2012)

#### 2.5.3.3.3 O COMANDO FOR

O comando *for* é responsável por executar *loops* no Python. Esse comando itera sobre os itens de uma sequência.

O comportamento do comando *for* do Python é diferente da maioria das linguagens de programação. Ao invés de trabalhar com variáveis contadoras, o *for* faz iteração sobre os valores de uma sequência. Para os desenvolvedores que precisem fazer com que o comando *for* tenha um comportamento semelhante as demais linguagens, é necessário usá-lo em conjunto com a função *range()*.

Assim como no *while*, caso o *loop* chegue ao fim sem ser interrompido por um comando *break*, os comandos do bloco de código do *else* serão executados, sendo este opcional.

Na Figura 16 é mostrada a sintaxe para o comando *for* do Python.



```

for <referência> in <sequência>:
    <bloco de código>
else:
    #opcional
    <bloco de código>

```

Figura 16 - Sintaxe do comando *for* em Python

Fonte: PRÓPRIA (2012)

Na Figura 17 segue alguns exemplos da utilização do comando *for*.

```

1  >>> for i in range(5):
2      print i,
3      else:
4      print "fim."
5  0 1 2 3 4 fim.
6
7  >>> a = ['gato', 'cachorro', 'cavalo']
8  >>> for x in a:
9      print x,
10 gato cachorro cavalo

```

Figura 17 - Exemplo da utilização do comando *for*

Fonte: PRÓPRIA (2012)

### 2.5.3.4 FUNÇÕES

As funções são um dispositivo de estruturação de programas, elas agrupam um conjunto de instruções de modo que elas possam ser executadas mais de uma vez em um programa. As funções também nos permitem especificar parâmetros que servem como entradas (LUTZ e ASCHER, 2007).

Funções têm como objetivo principal dividir tarefas (bloco de códigos) em subtarefas. A função é chamada por seu nome e pode ter vários parâmetros de entrada e pode ou não retornar valor.

Uma função de Python é definida com a palavra “*def*” seguida do nome da função e seus parâmetros de entrada entre (). O retorno de valores é por meio da expressão “*return*” seguida do valor.

Segundo Borges (2009) no Python, as funções:

- Podem retornar ou não objetos;

- Aceitam *strings* de documentação (*Doc Strings*);
- Aceitam parâmetros opcionais (defaults). Caso o parâmetro não seja passado o valor será igual ao default definido na função;
- Aceitam que os parâmetros sejam passados com nome. Neste caso, a ordem em que os parâmetros foram passados não importa;
- Tem *namespace* próprio (escopo local) e por isso podem ofuscar definições de escopo global;
- Podem ter suas propriedades modificadas

Na Figura 18 é mostrada a sintaxe das funções em Python:

```
def func(parametro1, parametro2=padrao):  
    """Doc String  
    """  
    <bloco de código>  
    return valor
```

Figura 18 - Sintaxe de funções em Python

Fonte: PRÓPRIA (2012)

#### 2.5.3.5 CLASSES PYTHON

O Python é uma linguagem orientada a objetos, logo, é possível definir novas classes com métodos e atributos, herdar classes já existentes e sobrescrever métodos. No Python a palavra “*class*” é usada para definir uma classe. As classes são acessadas da mesma forma que executa uma função. Seus métodos e atributos são acessados pelo nome da instancia seguido de um “.” mais o nome do método/atributo.

Na Figura 19 é mostrada a sintaxe das classes em Python.

```

class classe[(nomeClassePai1, nomeClassePai2,...)]:
    atributo = None
    def metodo(self):
        return valor

instancia = Classe()
instancia.metodo()
instancia.atributo

```

Figura 19 - Sintaxe de classes em Python

Fonte: PRÓPRIA (2012)

## 2.6 DJANGO

Django é um *framework* para desenvolvimento rápido para *web*, escrito em Python, que utiliza o padrão MVC (*Model-View-Controller*). Foi criado originalmente como sistema para gerenciar um *site* jornalístico na cidade de Lawrence, no estado do Kansas, nos Estados Unidos da América, pelos programadores Adrian Holovaty e Simon Willison, em 2003 (HOLOVATY e MOSS, 2009). Acabou se tornando um projeto de código aberto e foi publicado sob a licença BSD em 2005.

O Django foi desenvolvido com o objetivo de agilizar o desenvolvimento de *sites* dinâmicos, para que fosse possível desenvolver páginas ou aplicações com a qualidade e a velocidade exigidas em uma redação de jornal.

Os *sites* jornalísticos exigiam que recursos fossem adicionados e aplicações inteiras fossem desenvolvidas no curto período de tempo, em alguns casos até em poucos dias ou até horas de antecedência. Visto a necessidade de criar aplicações em curto espaço de tempo os programadores Holovaty e Willison desenvolveram um *framework* para reduzir o tempo de desenvolvimento dos *sites*, sendo a única maneira de conseguirem desenvolver aplicações sustentáveis em prazos extremos (HOLOVATY e MOSS, 2009).

Segundo Holovaty e Moss (2009) uma questão importante de ressaltar é como as origens do Django moldaram a cultura da sua comunidade de código aberto. Como o Django foi extraído do código de problemas reais, ao invés de ser um exercício acadêmico ou um produto comercial, ele é fortemente focado na resolução de problemas do desenvolvimento *web* que os programadores Django enfrentaram e continuam a enfrentar.

O Django possui uma série de características que o destacam dos demais *frameworks* para desenvolvimento *web*, entre elas estão (SANTANA e GALESI, 2010) e (HOLOVATY, 2009):

- Aplicações desenvolvidas com o Django seguem a arquitetura MVC;
- Automatização de algumas tarefas do processo de desenvolvimento;
- Redução do tempo necessário para a construção da aplicação;
- Mantém o desenvolvedor com o foco nas particularidades do negócio da aplicação;
- Possui uma forma de mapear as *Uniform Resource Locators* (URLs) requisitadas para as funções especificadas pelo desenvolvedor e possibilitando o uso de URLs amigáveis ao usuário;
- O *framework* possibilita a criação da área administrativa de maneira quase automática, possibilitando a administração dos objetos de negócio da aplicação;
- O Django facilita a internacionalização da aplicação, permitindo que ele funcione em diversos idiomas de maneira correta;
- Possui um poderoso *Object-Relational Mapping* (ORM) - Mapeador Objeto-Relacional;
- Possui um gerenciamento de autenticação de usuários e controle de permissões.

Os desenvolvedores do Django, o apelidaram de "O framework web para perfeccionistas com prazos". O Django apresenta um *design* limpo e a adesão de algumas práticas padrão para o desenvolvimento rápido (DJANGOPROJECT, 2012).

O Django, como a maioria dos frameworks, permite que o desenvolvedor siga o modelo de projetar e estruturar o código e beneficiar-se com isso, ou permite que seja criado um modelo próprio. Isso significa que se o desenvolvedor seguir determinadas convenções na maneira de estruturar seu código, não precisará ficar configurando características específicas do projeto (MOORE; BUDD; WRIGHT, 2007).

Logo a seguir serão destacadas as principais filosofias utilizadas pelos desenvolvedores na criação do *framework* Django.

- Fraco acoplamento: o Django tem o objetivo fundamental de fazer com que cada elemento da sua pilha de camadas seja independente. A pilha do Django deve ter um baixo acoplamento e uma alta coesão. Isso quer dizer, que as várias camadas do framework não devem "conhecer" sobre as demais, a menos que seja necessário. Por exemplo, o sistema de *template* não deve saber nada sobre o

processador de requisições, a camada de banco de dados não deve saber como será a exibição dos dados e o sistema de visões não deve se importar com qual o sistema de *template* o programador utilizará. Isso permite que várias peças da pilha de camadas possam ser trocadas, sem a necessidade de uma grande reescrita de outras partes. Isso faz com que o código seja mais fácil de manter. Em suma, quanto menor o acoplamento entre as partes, menor será o impacto de mudanças no projeto;

- Menos código: as aplicações em Django devem usar o mínimo de código possível. Como o Django é implementado em Python, é possível tirar o máximo da capacidade do Python para executar uma grande variedade de funcionalidades com poucas linhas de código. Além disso, o Django tenta diminuir a quantidade de código a ser produzido, fornecendo código e herança de *templates* para funções comuns;
- Desenvolvimento rápido: o ponto forte de um *framework* é proporcionar o desenvolvimento rápido, e com o Django isso não é diferente. Ele deve permitir o desenvolvimento rápido e robusto de *sites* ou aplicações *webs*;
- Não se repita (*Don't Repeat Yourself*, DRY): cada conceito distinto deve existir em exatamente um lugar. A existência de redundância é ruim para os projetos. Já a normalização é um ponto desejado. O Django, dentro do possível, tenta reduzir ao máximo o número de informações;
- Explícito é melhor que implícito: o Django não deve fazer muita mágica. Isto é, o código deve ser compacto, e não deve depender de truques, permitindo que desenvolvedores iniciantes consigam compreender;
- Consistência: o *framework* deve ser consistente em todas as camadas. Sendo assim, todas as camadas devem gerar resultados esperados;

A lista completa do conjunto de filosofias do Django está disponível no endereço <https://docs.djangoproject.com/en/1.3/misc/design-philosophies/>.

### 2.6.1 MTV

O Django usa o paradigma MVC (Modelo, Visão e Controlador) para o desenvolvimento de aplicações, mas por convenção, os idealizadores do Django optaram por utilizar a sigla MTV (*Model-Template-View*) para se referir as camadas do padrão MVC. Embora o Django não obrigue o desenvolver a seguir o padrão MVC, é altamente recomendada essa prática, uma vez que ela permite que sua aplicação seja modularizada e proporciona o desacoplamento da camada de negócio da camada de apresentação dos dados.

De maneira geral, a aplicação em Django é dividida da seguinte forma:

- O modelo (*Model*) é usado para definir e gerenciar a representação dos dados. Ele é uma representação detalhada da informação que a aplicação opera. No Django, a classe *Model* é responsável pela definição dos objetos;
- A visão (*View*) apresenta o modelo num formato adequado ao usuário e pode possuir diferentes visões para um mesmo modelo, com diferentes propósitos. No Django, o sistema de *templates* fica responsável pela apresentação da informação;
- O controlador (*Controller*) recebe a entrada de dados e retorna uma resposta. Responsável pelo acesso aos dados e pela manipulação dos mesmos. As funções como validação e filtragem ficam a cargo dessa camada. No Django, as funções do controlador são divididas entre dois componentes, o *URL Dispatcher* e as *Views*.

#### 2.6.1.1 MODELS

Um modelo é a fonte única e definitiva de dados sobre os dados, pois ele possui informações sobre os próprios dados da regra de negócio. Ele contém os campos e comportamentos básicos dos dados que a aplicação está manipulando. Os modelos de dados determinam as tabelas e as regras de negócio utilizadas pela aplicação. Geralmente, cada modelo é mapeado para uma única tabela no banco de dados. O Django cria as tabelas de modo transparente, de acordo com as definições das classes. Essas classes são denominadas

como classes *Model* e ficam armazenadas em um arquivo chamado `models.py` dentro do diretório de cada aplicação.

Informações sobre o modelo do Django:

- Cada modelo é uma classe Python que herda `django.db.models.Model`;
- Cada atributo do modelo representa uma coluna do banco de dados;
- Após a definição do modelo o Django lhe dá uma API de acesso ao banco de dados gerada automaticamente, onde é possível criar, receber, atualizar e deletar objetos. A utilização da API será explicada em uma seção 3.3.5 desse trabalho.

### 2.6.1.2 TEMPLATES

O Django fornece um sistema de *templates* para geração de páginas. Esse sistema permite que variáveis de um arquivo com marcações especiais sejam substituídas por valores gerados pela função *View*. Outra função realizada pelo sistema de *templates* é a de possibilitar a utilização de estruturas de controle, que o transformam em uma espécie de linguagem de programação. Uma das vantagens desse sistema é a separação da lógica de negócios da lógica de apresentação dos dados. Isso ocorre porque toda a parte HTML, CSS e JavaScript ficam dentro dos *templates* deixando as classes *Model* e as funções *View* responsáveis pelo processamento dos dados. O *template* é simplesmente um arquivo de texto. Podendo ser gerado por qualquer formato baseado em texto (HTML, XML, CSV, etc). (DJANGOPROJECT, 2011)

### 2.6.1.3 VIEWS

Uma função *View* é uma função Python que recebe uma requisição e retorna uma resposta. A resposta pode ser o conteúdo HTML de uma página, um redirecionamento, uma página de erro ou um documento qualquer.

O Django não exige que o arquivo que contém as *Views* fique em um diretório estabelecido, ele pode ser colocado em qualquer lugar do projeto.

As *Views* do Django desempenham o papel de *Controller*, de acordo com o paradigma MVC, juntamente com o componente *URL Dispatcher*, com que divide suas tarefas de controlador.

O *URL Dispatcher* é o componente do Django responsável por analisar as requisições. Esse componente é responsável por verificar o endereço da requisição e apontar qual será a *View* invocada para processar a resposta.

O Django utiliza expressões regulares para associar as *URLs* que acessam as funções na *View*. Expressões regulares são pequenas expressões que especificam padrões de reconhecimento texto. Então, é possível usá-las para reconhecer as características de uma *URL* e tratá-la de maneira adequada. O QUADRO 3 apresenta alguns exemplos de expressões regulares.

QUADRO 3  
Expressões regulares

(Continua)

Caractere	Significado
\$	Casa com fim da string.
^	Casa com início da string.
.	Casa com qualquer coisa.
( e )	Marcam um grupo. Um grupo é um conjunto de caracteres casando em conjunto. Grupos podem ganhar nomes, o que ajuda no processamento das expressões regulares. Pode-se dar um nome para um grupo colocando-se ?P<nome> logo depois do abre-parênteses, seguido pela expressão regular do grupo.
+, *, ?, { e }	Marcam quantidade: “+” indica zero ou mais caracteres, “?” indica zero ou um caractere, e “{ }” indicam uma quantidade específica de caracteres. Sempre se referem ao grupo, classe ou elemento à esquerda na expressão regular.
[ e ]	Marcam uma classe. Exemplo, [0-9] casa com os números entre 0 e 9, [aeiou] casa com vogais.
\d, \D	Casam com dígitos (\d) ou com o que não for dígito (\D).
\s, \S	Casam com caracteres de espaço em branco (\s) ou com o que não for espaço em branco (\S).
\w, \W	Casam com caracteres alfanuméricos (\w) ou com o que não for alfanumérico (\W).



**QUADRO 3**  
Expressões regulares

(Continuação)

<b>Caractere</b>	<b>Significado</b>
	Operador OU. Exemplo: (a b) casa com ‘a’ ou ‘b’.
\	Caractere de escape. Por exemplo, se quisermos casar algum dos caracteres especiais. Exemplo: \+ casa ou caractere ‘+’.

Fonte: SANTANA E GALESI (2010)

### 2.6.2 ORM

ORM é uma técnica de desenvolvimento utilizada para reduzir a resistência da programação orientada a objetos utilizando bancos de dados relacionais.

A maior parte das aplicações *web* utiliza banco de dados relacional para o armazenamento das suas informações. Geralmente, os bancos de dados utilizam a linguagem SQL para manipular os dados. Porém, mesmo sendo padrão, a linguagem SQL possui suas particularidades em cada servidor de banco de dados, já que eles não seguem totalmente a especificação da linguagem padrão. Além disso, algumas funcionalidades só existem em determinados sistema de banco de dados ou funcionam de maneira distinta dos demais. Isso gera uma grande dificuldade para os desenvolvedores conseguirem produzir aplicações que funcionem em diversos bancos de dados (SANTANA;GALESI, 2010).

O Django auxilia nessa tarefa de desenvolver aplicações portáteis, que funcionem em diversos bancos de dados, fornecendo uma ferramenta de ORM, que possibilita ao desenvolvedor manipular os dados sem a necessidade escrever código na linguagem SQL e sem se preocupar com os detalhes de funcionamento do sistema de banco de dados. O ORM possibilita o desenvolvimento de *software* utilizando o paradigma do desenvolvimento orientado a objetos (utilizando classes, atributos e métodos) e fazendo toda a interpretação para o modelo relacional automaticamente.

Isso permite que a aplicação seja criada de maneira independente da tecnologia utilizada pelo banco de dados. Facilitando a migração da aplicação para outros sistemas de banco de dados.

O *framework* Django utiliza os modelos de dados para representar uma tabela do banco de dados. Logo, uma instância desta classe representa um registro do banco de dados.

### 2.6.3 INTERFACE DE ADMINISTRAÇÃO

A área administrativa é uma interface onde é possível que o usuário altere o conteúdo das páginas de seu *site* ou aplicação. Essa interface é muito utilizada em aplicações cujo objetivo é apresentar conteúdos, como os portais, blogs, *sites* institucionais, dentre outros. Geralmente as ações executadas nessas interfaces são parecidas em todas as aplicações *web*, possibilitando o usuário alterar, buscar, adicionar e excluir um registro.

Com o objetivo de facilitar o desenvolvimento destas interfaces de manutenção de conteúdo, o Django oferece uma interface de administração, criada automaticamente de acordo com os modelos de dados, oferecendo um mecanismo poderoso para gerenciar os dados de uma aplicação, sem a necessidade de desenvolver cada uma das interfaces de cadastro de determinados tipos de objetos. Esse é um dos principais diferenciais do Django em relação a outros *frameworks* de desenvolvimento *web* (SANTANA; GALESI, 2010).

## 2.7 GEODJANGO

GeoDjango é uma extensão do *framework* Django, cujo objetivo é fornecer suporte à construção de aplicações *web* geoespaciais e aproveitar o poder dos dados espaciais. Com ele é possível utilizar um banco de dados com extensões geográficas, como o PostGIS, com o PostgreSQL, além de permitir fazer consultas de distância, perímetro, áreas, entre outras.

Alguns recursos do GeoDjango incluem [DJANGOPROJECT, 2011]:

- Campos do modelo Django para geometrias OGC, o modelo do Django é estendido para aceitar tipos de dados geoespaciais;
- Extensões para o ORM do Django para consultar e manipular dados espaciais;
- Edição de campos geométricos dentro da área administrativa nativa do Django.

O GeoDjango está incorporado ao Django, então, as instruções de instalação são referentes à do Django. Os bancos de dados espaciais suportados são: PostgreSQL (com PostGIS), MySQL, Oracle e o SQLite (com SpatiaLite).

Para utilizar o GeoDjango é necessário instalar algumas bibliotecas geoespaciais e a escolha das bibliotecas depende do banco de dados espacial utilizado. No QUADRO 4 são listadas as bibliotecas necessárias e as versões suportadas de cada banco de dados suportado pelo GeoDjango.

#### QUADRO 4

##### Requisitos de bibliotecas dos bancos de dados espaciais

Banco de dados	Bibliotecas	Versões suportadas
PostgreSQL (PostGIS)	GEOS, PROJ.4, PostGIS	8.1+
MySQL	GEOS	5.x
Oracle	GEOS	10.2, 11
SQLite (SpatiaLite)	GEOS, GDAL, PROJ.4, SpatiaLite	3.6.+

Fonte: DJANGOPROJECT (2012)

Informações adicionais sobre as bibliotecas e os procedimentos para a instalação estão disponíveis em <http://docs.djangoproject.com/en/dev/ref/contrib/gis/install/#geodjango-installation>.

Para apoiar os dados geoespaciais, o GeoDjango fornece algumas extensões do modelo de dados padrão do Django, no QUADRO 5 são mostrados os novos tipos de dados adicionados ao *framework*.

#### QUADRO 5

##### Listagem dos novos tipos introduzidos pelo GeoDjango ao modelo padrão do Django

(Continua)

Campo	Descrição
GeometryField	Campo genérico que pode conter qualquer uma das geometrias abaixo
PointField	Campo que irá conter a informação de um ponto
LineStringField	Campo que irá conter a informação de uma linha
PolygonField	Campo que irá conter a informação de um polígono
MultiPointField	Campo que irá conter a informação de múltiplos pontos

## QUADRO 5

Listagem dos novos tipos introduzidos pelo GeoDjango ao modelo padrão do Django

(Continuação)

<b>Campo</b>	<b>Descrição</b>
MultiLineStringField	Campo que irá conter a informação de múltiplas linhas
MultiPolygonField	Campo que irá conter a informação de múltiplos polígonos
GeometryCollectionField	Campo que irá conter uma coleção de linhas, pontos e polígonos.

Fonte: DJANGOPROJECT (2012)

## 2.8 GOOGLE MAPS

O Google Maps é um serviço gratuito lançado pela empresa Google em fevereiro de 2005, usado para realizar pesquisas, visualizar imagens de satélite e mapas. O Google Maps oferece recursos para a criação de um mapa completo, permitindo marcar locais, adicionar imagens e vídeos e compartilhar esse conteúdo na *web* (AZEVEDO, 2008).

## 2.8.1 API GOOGLE MAPS

A API do Google Maps é uma interface de desenvolvimento para aplicações baseadas no Google Maps, ela é formada por um conjunto simples de classes JavaScript, que rodam diretamente dentro de uma página HTML. Todas as funcionalidades de um mapa do Google se baseiam na simples premissa de um objeto construído com JavaScript incorporado em uma página da *Web*. Isso possibilita criar aplicativos inovadores de georreferenciamento para aplicativos *web*. A API auxilia a integrar mapas e adicionar geocodificações aos *sites*, possibilitando que aplicações que possuam dados georreferenciados sejam facilmente mostradas em qualquer navegador.

A API é simples, consiste basicamente em um pequeno grupo de classes JavaScript que fornecem uma interface para que o usuário possa desenvolver aplicações para

exibir mapas, realizar consultas por endereços, realizar operações de *zoom*, arraste clique, adicionar pontos de referência ou descrever algum ponto no mapa, dentre várias outras possibilidades (GOOGLE DEVELOPERS, 2012).

Segundo o Google Developers (2012) os conceitos relacionados à API do Google Maps estão divididos da seguinte forma:

- Objetos básicos;
- Eventos;
- Controles de mapa;
- Sobreposições;
- Serviços.

#### **2.8.1.1 OBJETOS BÁSICOS**

Segundo o Google Developers (2012), o elemento essencial de qualquer aplicativo da API do Google Maps é o “próprio” mapa.

Na Figura 20 é mostrado um exemplo para a criação de um mapa utilizando a API. No exemplo, o mapa é centralizado na Universidade Estadual de Montes Claros (UNIMONTES), com o nível de *zoom* marcado com o valor 17 e utilizando a visualização de imagens de satélite do Google Earth.

```

1 <html>
2 <head>
3 <meta name="viewport" content="initial-scale=1.0, user-scalable=no" />
4 <script type="text/javascript"
5   src="https://maps.google.com/maps/api/js?sensor=false"><!--Incluído a API Javascript
   do Google Maps com sensor de GPS desativado -->
6 </script>
7 <script type="text/javascript">
8   function initialize() {
9     var latlng = new google.maps.LatLng(-16.71986, -43.88017); //objeto com latitude e longitude
10    var myOptions = { //Objeto com as propriedades do mapa
11      zoom: 17, //nível de zoom
12      center: latlng, //centralização na latitude e longitude do objeto latlng
13      mapTypeId: google.maps.MapTypeId.SATELLITE //tipo do mapa
14    };
15    var map = new google.maps.Map(document.getElementById("map_canvas"),
16      myOptions); //Objeto do mapa que recebe como parâmetro o elemento HTML onde o mapa será colocado e as propriedades do objeto myOptions
17  }
18 </script>
19 </head>
20 <body onload="initialize()"> <!--
21   <div id="map_canvas" style="width:100%; height:100%"></div>
22 </body>
23 </html>

```

Figura 20 - Código para criação de um mapa

Fonte: PRÓPRIA (2012)

Na Figura 21 é mostrado o resultado da Figura 20.0



Figura 21 - Exemplo de um mapa básico.

Fonte: PRÓPRIA (2012)

Além da visualização de imagens de satélite, mostrado no exemplo anterior, existem outros tipos de mapas, os tipos disponíveis na API do Google Maps são:

- `MapTypeId.ROADMAP` exibe a visualização de mapa padrão;
- `MapTypeId.SATELLITE` exibe imagens de satélite do Google Earth;
- `MapTypeId.HYBRID` exibe uma mistura entre as visualizações normal e de satélite;
- `MapTypeId.TERRAIN` exibe um mapa físico com base nas informações do terreno.

### 2.8.1.2 EVENTOS

O JavaScript no navegador é responsável por responder às interações realizadas pelos usuários. Cada interação do usuário gera um evento específico e uma função específica pode ser configurada para cada tipo de evento disparado. Todas as vezes que o evento for disparado a sua função será executada.

Na API do Google Maps os eventos são divididos em duas categorias (GOOGLE DEVELOPERS, 2012):

- Eventos de usuário (como eventos de "clique" do *mouse*);
- Evento de alteração de estado. Sempre que a propriedade de um objeto mudar, a API disparará um evento sobre essa alteração, a nomeação desses eventos segue a seguinte convenção *property\_changed*.

Segundo o Google Developers (2012) para receber notificações é usado o manipulador de evento `addListener()`. Essa função usa um objeto, o nome do evento a ser ouvido e função ser chamada quando ocorre o evento especificado.

Na Figura 22 é mostrado um exemplo que contém um evento de usuário e um evento de alteração de estado. No marcador foi anexado um manipulador de evento, que amplia o mapa quando clicado. O outro manipulador de evento foi anexado ao mapa para receber notificações quando a propriedade “*zoom*” for modificada, quando a propriedade for alterada ele executará uma função que centraliza o mapa na coordenada que se encontra o marcador.

```

1 <html>
2 <head>
3 <meta name="viewport" content="initial-scale=1.0, user-scalable=no" />
4 <script type="text/javascript"
5   src="https://maps.google.com/maps/api/js?sensor=false">
6 </script>
7 <script type="text/javascript">
8   var map;
9   function initialize() {
10     var myLatLng = new google.maps.LatLng(-16.71986, -43.88017);
11     var myOptions = {
12       zoom: 4,
13       center: myLatLng,
14       mapTypeId: google.maps.MapTypeId.ROADMAP
15     }
16     map = new google.maps.Map(document.getElementById("map_canvas"), myOptions);
17     //manipulador de evento adicionado ao mapa para a propriedade zoom_changed
18     google.maps.event.addListener(map, 'zoom_changed', function() {
19       setTimeout(moveToCenter, 3000);
20     });
21     //definido um marcador para o mapa
22     var marker = new google.maps.Marker({
23       position: myLatLng, //posição do marcador
24       map: map, //o mapa que o marcador irá aparecer
25       title:"Hello World!" //título do marcador
26     });
27     google.maps.event.addListener(marker, 'click', function(event) { //manipulador de evento para
o marcador, disparado quando ocorre o clique sobre ele.
28       map.setZoom(18);
29     });
30   }
31
32   function moveToCenter() {
33     var latlng = new google.maps.LatLng(-16.71986, -43.88017);
34     map.setCenter(latlng);
35   }
36 </script>
37 </head>
38 <body onload="initialize()">
39   <div id="map_canvas" style="width:100%; height:100%"></div>
40 </body>
41 </html>

```

Figura 22 - Exemplo de eventos do Google Maps

Fonte: PRÓPRIA (2012)

A lista completa dos eventos da API está disponível no endereço <https://developers.google.com/maps/documentation/javascript/reference?hl=pt-BR>.



### 2.8.1.3 CONTROLES

O Google Maps possui elementos de interface que possibilitam a interação do usuário. Esses elementos são denominados de controles.

A API permite que o desenvolvedor inclua variações desses controles, caso contrário ela controla o comportamento de todos eles.

A API permite personalizar a interface do mapa adicionando, removendo ou modificando a interface de usuário ou dos controles. Alguns controles aparecem no mapa por padrão, já outros, só ficam visíveis quando solicitados. Para alterar a visibilidade dos controles é necessário alterar os campos do objeto *Map Options*, marcando como *true* para ficarem visíveis ou como *false* para ficarem ocultos (GOOGLE DEVELOPERS, 2012).

O exemplo mostrado na Figura 23 oculta os controles de navegação (*zoom* e *pan*) e exibe o controle de escala.

```
1 <html>
2 <head>
3 <meta name="viewport" content="initial-scale=1.0, user-scalable=no" />
4 <script type="text/javascript"
5   src="https://maps.google.com/maps/api/js?sensor=false">
6 </script>
7 <script type="text/javascript">
8   var map;
9   function initialize() {
10    var myOptions = {
11      zoom: 12,
12      center: new google.maps.LatLng(-16.71986, -43.88017),
13      panControl: false, //oculta as propriedades de pan
14      zoomControl: false, //oculta as propriedades de zoom
15      scaleControl: true, //exibe o controle de escala
16      mapTypeId: google.maps.MapTypeId.ROADMAP
17    }
18    var map = new google.maps.Map(document.getElementById("map_canvas"),
19      myOptions);
20  }
21 </script>
22 </head>
23 <body onload="initialize()">
24   <div id="map_canvas" style="width:100%; height:100%"></div>
25 </body>
26 </html>
```

Figura 23 - Oculta e exibe controles no mapa

Fonte: PRÓPRIA (2012)

Na Figura 24 é mostrado o resultado da Figura 23.

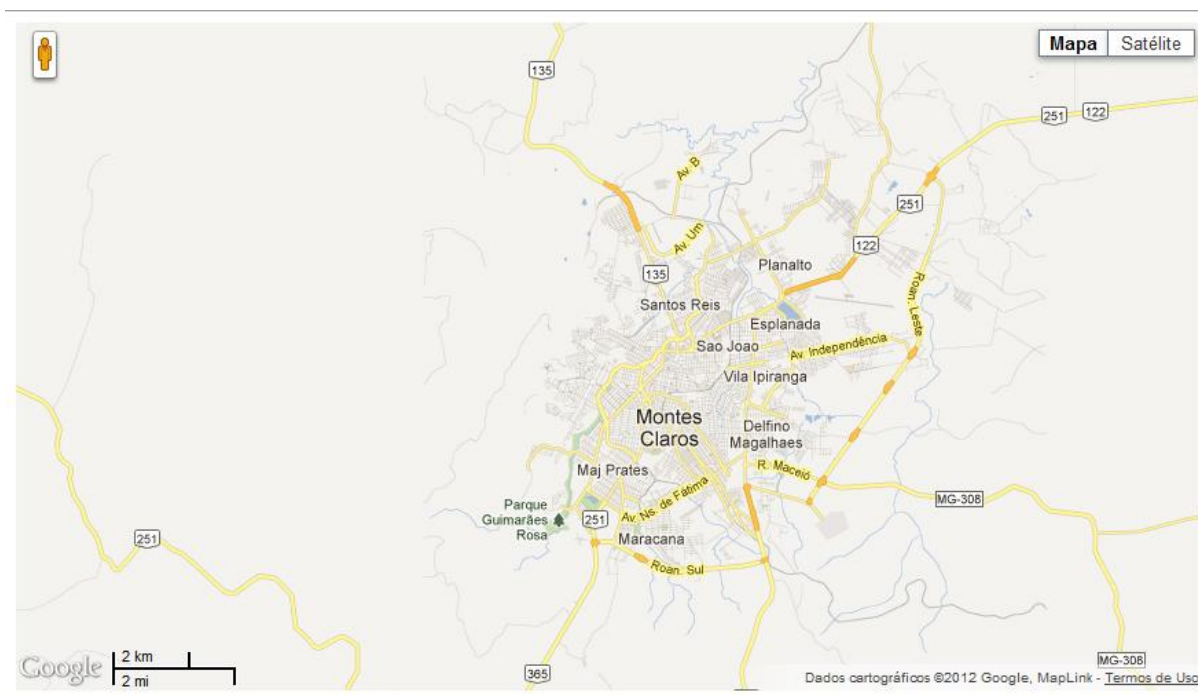


Figura 24 - Mapa sem os controles de navegação e com controle escala ativado.

Fonte: PRÓPRIA (2012)

#### 2.8.1.4 SOBREPOSIÇÕES

Sobreposições são objetos no mapa presos a coordenadas de latitude e longitude, desta forma, eles se movimentam quando o mapa é arrastado ou ampliado. As sobreposições mostram os objetos que são adicionados ao mapa para indicar pontos, linhas, polígonos ou coleções de objetos (GOOGLE DEVELOPERS, 2012).

A API do Google Maps possui várias sobreposições, entre elas estão (GOOGLE DEVELOPERS, 2012):

- Locais ou pontos individuais que são exibidos no mapa por meio de marcadores. Os marcadores podem mostrar imagens personalizadas, sendo assim, chamados de "ícones". Os marcadores e ícones são objetos do tipo *Marker*;
- As linhas são exibidas por meio de polilinhas, que representam uma sequência ordenada de locais. As linhas são objetos do tipo *Polyline*;
- As áreas sem forma definida são exibidas no mapa por meio de polígonos. Como as polilinhas, os polígonos também são sequências ordenadas de pontos. Diferente das polilinhas, os polígonos definem uma região delimitada por eles;

- As camadas do mapa podem ser apresentadas usando tipos de mapa de sobreposição. O desenvolvedor pode produzir seu próprio conjunto de blocos criando tipos de mapa customizados que substituem ou são exibidos sobre os conjuntos de blocos de mapas base na forma de sobreposições;
- A janela de informações exibe conteúdo dentro de um balão *pop-up* por cima de um mapa em determinado ponto, sendo um tipo especial de sobreposição. Após ser criada, a janela de informações não é adicionada ao mapa. Para deixá-la visível, é necessário chamar o método *open()*, passando-o para o Mapa no qual será aberta e, opcionalmente, para o *Marker* com o qual será ancorada. Se nenhum marcador for passado como parâmetro, a janela de informações será aberta na sua propriedade *position*. As janelas de informações são do tipo *InfoWindow*;
- A API também permite que o desenvolvedor implemente suas próprias sobreposições customizadas, necessitando apenas que o desenvolvedor implemente a interface *OverlayView*.

Na Figura 25 e Figura 26 seguem exemplos de sobreposições de blocos, marcador e de janela de informações.

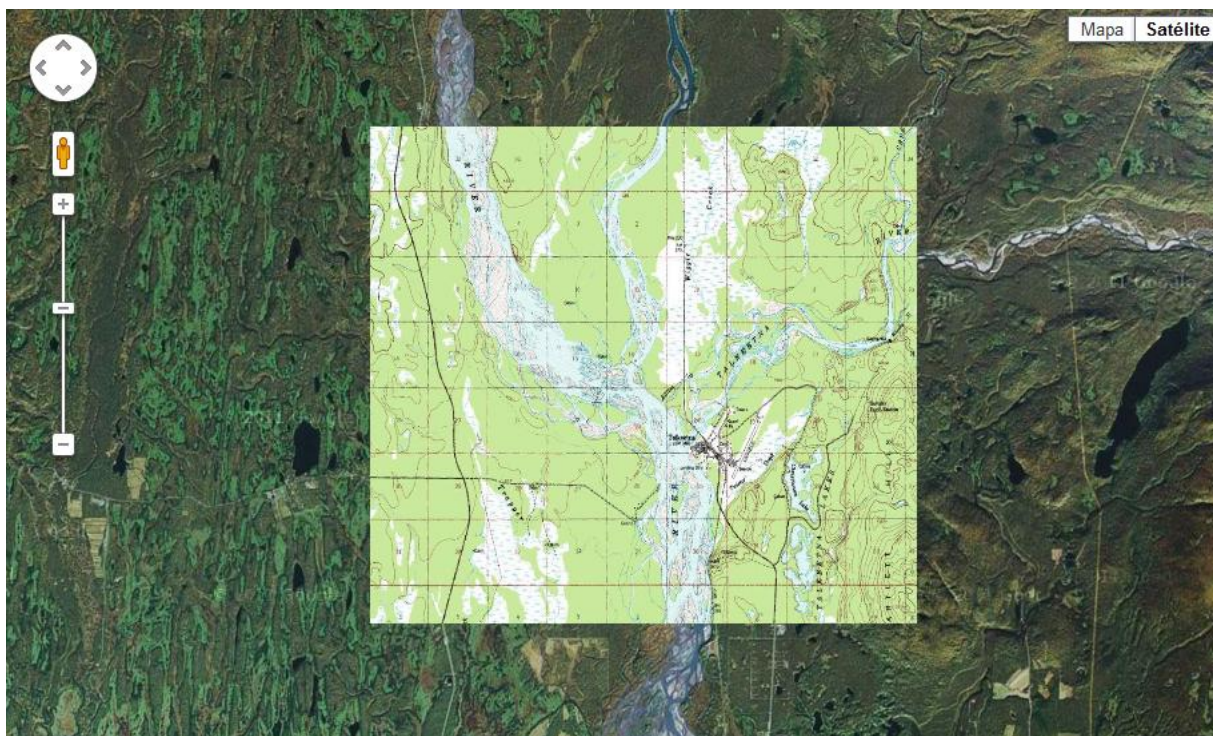


Figura 25 - Exemplo de sobreposição

Fonte: GOOGLE DEVELOPERS (2012)

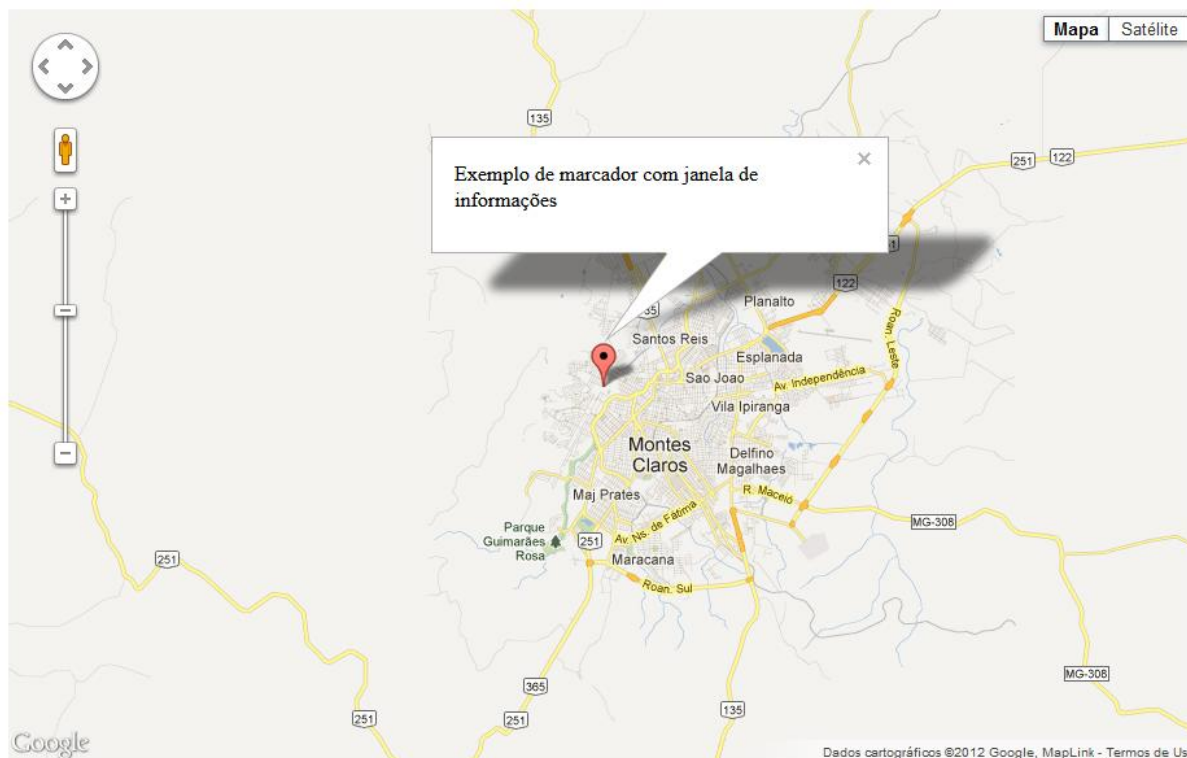


Figura 26 - Exemplo de marcador e de janela de informações

Fonte: PRÓPRIA (2012)

### 2.8.1.5 SERVIÇOS

A API do Google Maps permite que serviços externos solicitem e utilizem seus dados, para isso, ela oferece uma interface de serviços da *web* desenvolvidos para trabalharem em conjunto com um mapa. Dos serviços disponíveis na API do Google Maps, serão explicados nessa seção os serviços de: geocodificação, rotas e o *Street View*.

#### 2.8.1.5.1 GEOCODIFICAÇÃO

Geocodificação é ação de converter endereços em coordenadas geográficas, que são usados para posicionar o mapa ou adicionar marcadores. A API provê uma classe de geocodificação para geocodificar endereços dinamicamente, a partir da entrada do usuário.

Isso permite que usuários pesquisem regiões, locais e cidades a partir de endereços textuais (GOOGLE DEVELOPERS, 2012).

Na Figura 27 é mostrado um exemplo de geocodificação, buscando o endereço passado pelo usuário e adicionando um marcador nas coordenadas do endereço encontrado.

```

1  <html>
2  <head>
3  <meta name="viewport" content="initial-scale=1.0, user-scalable=no" />
4  <meta http-equiv="content-type" content="text/html; charset=UTF-8"/>
5  <script type="text/javascript"
   src="https://maps.googleapis.com/maps/api/js?sensor=false"></script>
6  <script type="text/javascript">
7    var geocoder;
8    var map;
9    function initialize() {
10     geocoder = new google.maps.Geocoder(); //objeto de geocodificação
11     var latlng = new google.maps.LatLng(-16.71986, -43.88017);
12     var myOptions = {
13       zoom: 12,
14       center: latlng,
15       mapTypeId: google.maps.MapTypeId.ROADMAP
16     }
17     map = new google.maps.Map(document.getElementById("map_canvas"), myOptions);
18   }
19   //função disparada quando o botão Buscar é clicado
20   function codeAddress() {
21     var address = document.getElementById("address").value; //lê o valor do campo texto
22     geocoder.geocode( { 'address': address}, function(results, status) { //passa a variável do
23     endereço como parâmetro
24     if (status == google.maps.GeocoderStatus.OK) {
25       map.setCenter(results[0].geometry.location); //centraliza o mapa no primeiro resultado
26       var marker = new google.maps.Marker({ //adicionado um marcador ao primeiro
27       resultado
28         map: map,
29         position: results[0].geometry.location
30       });
31     } else {
32       alert("Não foi encontrado nenhum resultado para essa pesquisa: " + status);
33     }
34   }
35   </script>
36   <body onload="initialize()">
37     <div>
38       <input id="address" type="text" value="">
39       <input type="button" value="Buscar" onclick="codeAddress()">
40     </div>
41     <div id="map_canvas" style="width: 100%; height: 500px;"></div>
42   </body>
43 </html>

```

Figura 27 - Exemplo de geocodificação a partir da entrada do usuário

Fonte: PRÓPRIA (2012)



Na Figura 28 é mostrado o resultado da Figura 27.

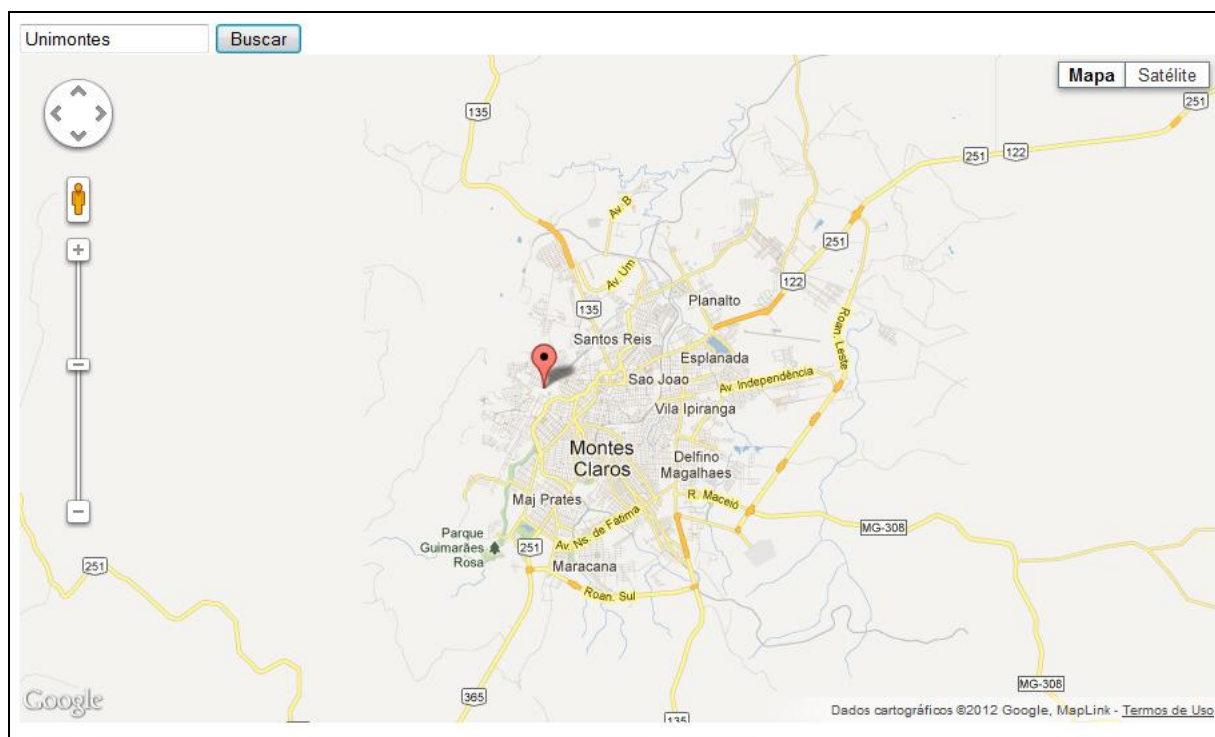


Figura 28 - Exemplo de geocodificação

Fonte: PRÓPRIA (2012)

### 2.8.1.5.2 ROTAS

Com o serviço de Rotas é possível calcular rotas, por distintos meios de transporte, utilizando o objeto *DirectionsService*. Esse objeto faz a comunicação com o serviço de rotas da API do Google Maps, que é responsável por receber as requisições de rota e devolver os resultados calculados. Os resultados de rotas podem ser manipulados manualmente ou pode ser renderizados pelo objeto *DirectionsRenderer*.

As rotas de origem e destino podem ser especificadas a partir de textos ou a partir de coordenadas geográficas.

A Figura 29 exibe a uma rota gerada a partir do ponto 1 (Unimontes, Montes Claros, MG) até o ponto 2 (Rua Dr. Santos, Montes Claros, MG).

```

1 <html>
2 <head>
3 <meta name="viewport" content="initial-scale=1.0, user-scalable=no"/>
4 <meta http-equiv="content-type" content="text/html; charset=UTF-8"/>
5 <script type="text/javascript"
6   src="https://maps.googleapis.com/maps/api/js?sensor=false"></script>
7 <script type="text/javascript">
8   var directionDisplay;
9   var directionsService = new google.maps.DirectionsService();
10   var map;
11   function initialize() {
12     directionsDisplay = new google.maps.DirectionsRenderer();
13     var centerMap = new google.maps.LatLng(-16.71986, -43.88017);
14     var myOptions = {
15       zoom:12,
16       mapTypeId: google.maps.MapTypeId.ROADMAP,
17       center: centerMap
18     }
19     map = new google.maps.Map(document.getElementById("map_canvas"), myOptions);
20     directionsDisplay.setMap(map); //seta qual o mapa o serviço de rota irá utilizar
21   }
22
23   function calcRoute() {
24     var start = document.getElementById("start").value;
25     var end = document.getElementById("end").value;
26     var request = { //objeto de rota
27       origin:start, //endereço de início da rota
28       destination:end, //endereço de fim da rota
29       travelMode: google.maps.DirectionsTravelMode.DRIVING //rota gerada para o modo
automotivo
30     };
31     directionsService.route(request, function(response, status) { //função de rotear
32       if (status == google.maps.DirectionsStatus.OK) {
33         directionsDisplay.setDirections(response); //faz a marcação da rota no mapa
34       }
35     });
36   }
37 </script>
38 </head>
39 <body onload="initialize()">
40 <div>
41   <label>Origem:</label></label><input type="text" id="start">
42   <label>Destino:</label></label><input type="text" id="end">
43   <input type="button" value="Gerar rota" onclick="calcRoute()">
44 </div>
45 <div id="map_canvas" style="width: 100%; height:100%"></div>
46 </body>
47 </html>

```

Figura 29 - Exemplo de rotas de dois pontos passados pelo usuário

Fonte: PRÓPRIA (2012)

Na Figura 30 é mostrado o resultado do código acima, após o usuário entrar com o endereço de origem e destino e clicar em Gerar rota.

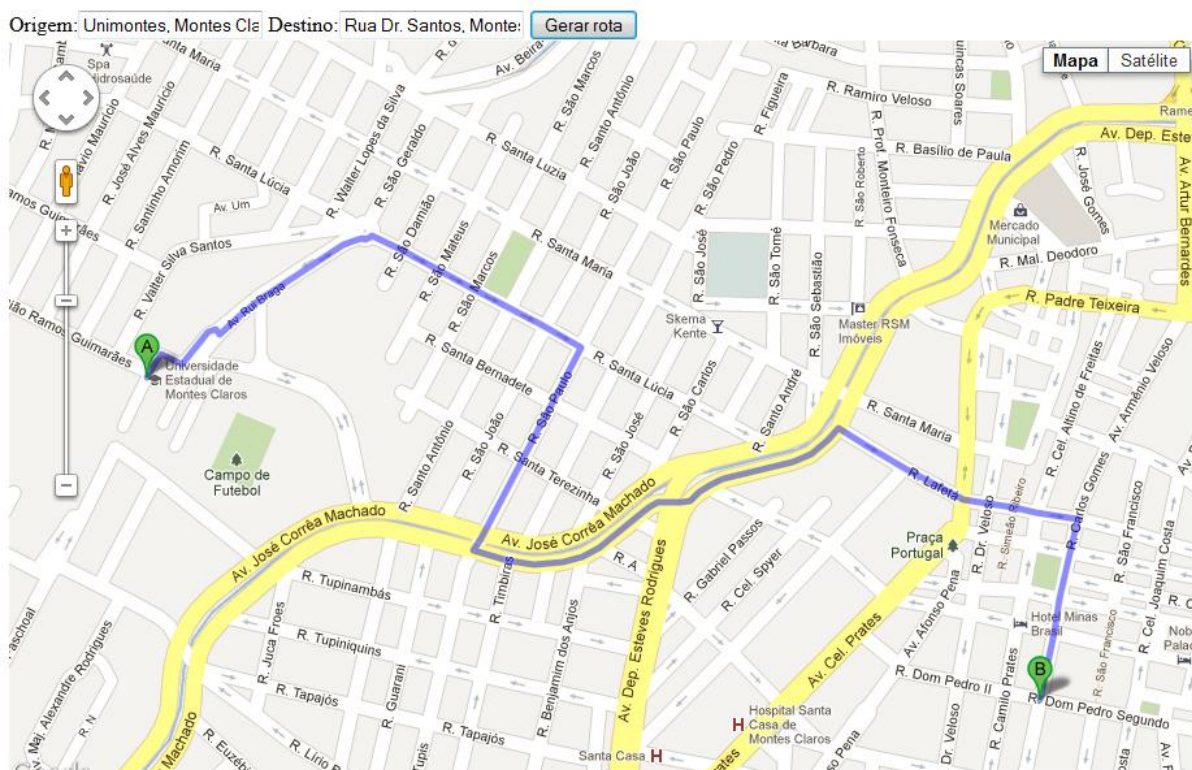


Figura 30 - Exemplo de rota gerada

Fonte: PRÓPRIA (2012)

### 2.8.1.5.3 STREET VIEW

O *Street View* do Google disponibiliza visualizações panorâmicas de 360 graus de ruas designadas em toda sua área de cobertura. A cobertura da *Street View* é a mesma do aplicativo Google Maps.

Por padrão, o *Street View* é ativado pelo controle *Pegman* (ícone de homem) que é exibido juntamente com os controles de navegação.

A Figura 31 exibe o código para exibição de um mapa com visualização pelo *Street View*. Nesse caso, a visualização do *Street View* foi habilitada manualmente pelo código, sem a necessidade de clicar no ícone de controle para ativá-la.



```

1 <html>
2 <head>
3 <meta http-equiv="content-type" content="text/html; charset=utf-8"/>
  <script src="https://maps.googleapis.com/maps/api/js?sensor=false"
4 type="text/javascript"></script>
5 <script type="text/javascript">
6   function initialize() {
7     var fenway = new google.maps.LatLng(-16.721924,-43.876508);
8     var mapOptions = {
9       center: fenway,
10      zoom: 14,
11      mapTypeId: google.maps.MapTypeId.ROADMAP
12    };
13    var map = new google.maps.Map(
14      document.getElementById("map_canvas"), mapOptions);
15    var panoramaOptions = { //objeto com as propriedades do mapa
16      position: fenway,
17      pov: {
18        heading: 290,
19        pitch: 0,
20        zoom: 2
21      }
22    };
23    var panorama = new
      google.maps.StreetViewPanorama(document.getElementById("map_canvas"),panoramaOptions);
//objeto da visão do Street View que recebe como parâmetro o elemento HTML onde o
mapa será colocado e as propriedades do objeto panoramaOptions
24    map.setStreetView(panorama); //muda a visão do mapa para a visão do Street View
25  }
26 </script>
27 </head>
28 <body onload="initialize()">
29   <div id="map_canvas" style="width: 100%; height: 500px"></div>
30 </body>
31 </html>

```

Figura 31 - Exemplo código para visualização do Street View

Fonte: PRÓPRIA (2012)

Na Figura 32 é mostrado o resultado da Figura 31.



Figura 32 - Exemplo de mapa na visualização do Street View

Fonte: PRÓPRIA (2012)

### 3 DESENVOLVIMENTO

Este trabalho tem como foco expor todos os passos e requisitos necessários para desenvolver uma aplicação geográfica utilizando o *framework* Django junto com a sua extensão para GIS, o Geodjango. Desta forma, foi definido que, para demonstração do trabalho, seria feito o desenvolvimento de uma aplicação de demonstração, cujo objetivo principal é mostrar a integração das tecnologias citadas no presente trabalho.

A partir dessa definição, o trabalho procedeu com o estudo das tecnologias utilizadas, bem como a definição das ferramentas e tecnologias que seriam utilizadas durante o desenvolvimento da aplicação, assim como as versões das mesmas.

Posteriormente, fez-se a preparação do ambiente de desenvolvimento, com a instalação da linguagem Python, da *IDE* de desenvolvimento, do *framework* Django, do servidor de banco de dados juntamente com a sua extensão espacial e das bibliotecas necessárias.

Por fim, o desenvolvimento da aplicação de demonstração.

#### 3.1 DEFINIÇÃO DAS TECNOLOGIAS

As tecnologias utilizadas nesse trabalho se dividem em três camadas: a interface, a servidor e o banco de dados. Cada uma dessas camadas possui o seu elemento principal, que junto com os elementos das demais camadas fornecem a estrutura básica para a construção de uma aplicação geográfica.

Na camada de interface foi escolhida a API do Google Maps, que é uma interface de desenvolvimento para aplicações baseadas no Google Maps, formada basicamente por classes JavaScript. A interação do usuário com o mapa acontece por meio dessa camada, e a partir dela, ele pode visualizar, adicionar ou consultar algum ponto de interesse do mapa.

Na camada de servidor foi escolhido o *framework* Django para atender às requisições da camada de interface. Ele é responsável por processar, analisar, validar e acessar o banco de dados quando necessário. O Django foi a ferramenta escolhida para a camada de

servidor, devido às suas características que estimulam o desenvolvimento ágil e da sua extensão GIS.

Na camada de banco de dados foi escolhido o PostgreSQL juntamente com a sua extensão espacial, o PostGIS. Segundo o DJANGOPROJECT (2012) o PostGIS é recomendado, porque é o banco de dados espacial aberto mais maduro, rico em recursos e compatível com o GeoDjango.

A Figura 33 apresenta a arquitetura dos componentes de uma aplicação geográfica, utilizando o GeoDjango, proposta por Bronn (2008).

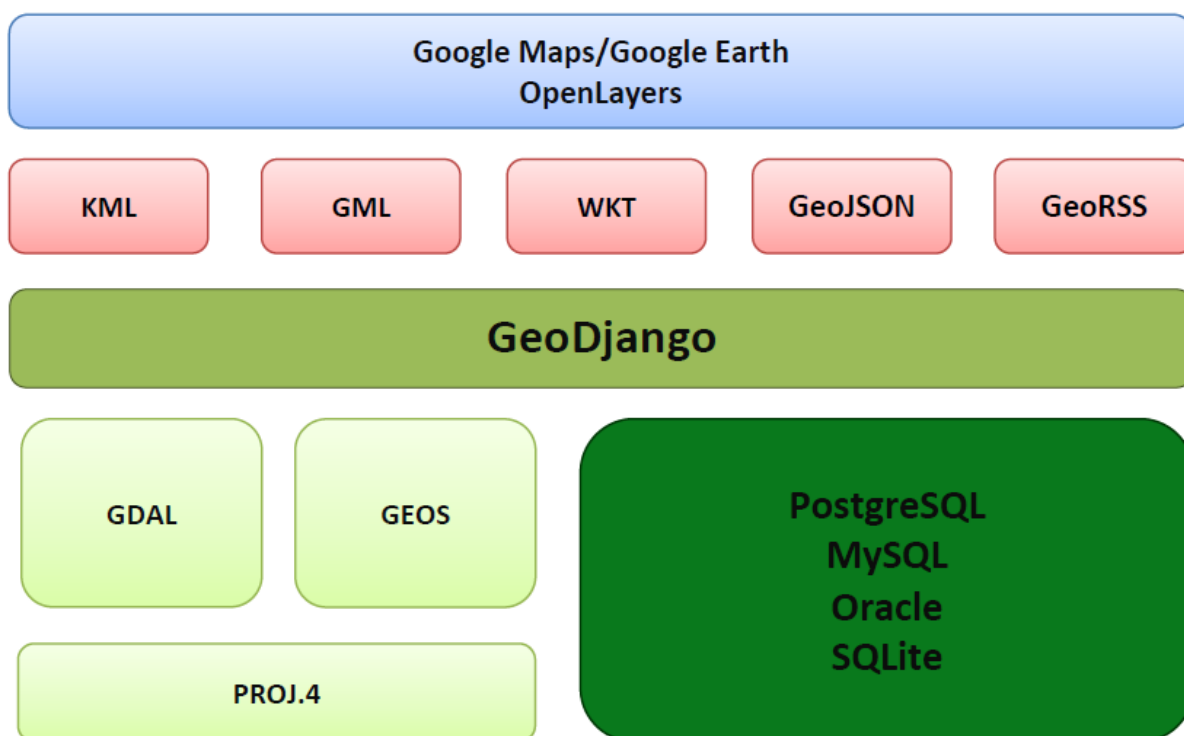


Figura 33 - Arquitetura de uma aplicação geográfica com GeoDjango.

Fonte: Adaptado de BRONN (2008)

## 3.2 PREPARAÇÃO DO AMBIENTE

Para a realização deste projeto foi utilizado a distribuição Linux Ubuntu 10.10, de 32 bits.

Para desenvolver a aplicação de demonstração é necessário a preparação do ambiente de desenvolvimento, com os programas e ferramentas necessárias. Segue a lista dos componentes necessários:

- API do Google Maps - versão 3;

- Interpretador Python - versão 2.6.6 (distribuída nativamente com o Ubuntu);
- Django - versão 1.3;
- PostgreSQL - versão 9.1.3;
- Extensão espacial PostGIS - versão 1.5.2;
- Editor de texto: IDE Eclipse com o *plug-in* PyDev ou o editor nativo do Ubuntu, o Gedit.

### 3.3 CONFIGURAÇÃO DA ARQUITECTURA PROPOSTA

Para desenvolver uma aplicação *web* no Django, independente de ser ou não uma aplicação geográfica, algumas configurações devem ser realizadas.

Nessa seção serão mostrados todos os procedimentos utilizados para instalar o *framework* Django e para configurar um projeto.

#### 3.3.1 INSTALAÇÃO DA FERRAMENTA

O único pré-requisito para instalar o Django é que o sistema possua a tecnologia Python. Para instalar o Python é necessário adquirir o arquivo de instalação no *site* oficial no endereço <http://www.python.org>. As distribuições Linux já disponibilizam o Python nativamente, podendo ser necessário atualizá-lo para uma versão mais recente.

Para instalar o Django, é necessário adquirir o arquivo de instalação, que pode ser encontrado no *site* oficial do projeto, no endereço <http://www.djangoproject.com/download/> ou no repositório oficial no Github, no endereço <http://github.com/django/django>.

Após adquirir o arquivo e descompactá-lo, basta executar o *script* `setup.py`, com o parâmetro *install*, para o Django ser instalado na máquina, conforme mostrado na Figura 34:

```
$ tar xfvz Django-install.tar.gz  
$ cd Django-install  
$ python setup.py install
```

Figura 34 - Comando para descompactar e instalar o Django

Fonte: PRÓPRIA (2012)

A referência oficial para a instalação do Django pode ser encontrada no endereço <https://docs.djangoproject.com/en/1.3/topics/install/>.

### 3.3.2 INICIANDO UM PROJETO

Quando o Django é instalado, ele cria um utilitário de linha de comando chamado de `django-admin.py`. Esse utilitário permite executar diversas tarefas de manutenção dos projetos que utilizam o *framework*. Uma dessas tarefas é a criação de um novo projeto com uma estrutura de arquivos e diretórios padrão, já preparada para ser usada.

Para iniciar um projeto é necessário executar o comando `django-admin.py` com o parâmetro *startproject* e o nome do projeto, conforme o exemplo na Figura 35:

```
$ django-admin.py startproject nome_do_projeto
```

Figura 35 - Comando para criar um novo projeto em Django

Fonte: PRÓPRIA (2012)

Após a execução do comando, o Django criou um diretório com o nome do projeto (`nome_do_projeto`) e dentro desse diretório, criou quatro arquivos que serão a estrutura básica do sistema/site. A seguir será mostrada uma descrição detalhada da função de cada arquivo criado pelo `django-admin.py`.

- `__init__.py`: arquivo vazio que indica ao Python que o diretório atual deve ser considerado como um pacote;
- `manage.py`: utilitário semelhante ao `django-admin.py`, utilizado para auxiliar na execução das tarefas específicas do projeto, como criar uma nova aplicação, validar ou sincronizar o projeto com o banco de dados.
- `settings.py`: arquivo com as configurações do projeto. É nesse arquivo que o Django busque informações de como conectar ao banco de dados, como enviar e-mail, quais os idiomas suportados pela aplicação, quais as aplicações utilizadas, etc;
- `urls.py`: esse arquivo é responsável pela definição das *URLs* do projeto, nele é mapeado qual *View* irá responder a uma certa requisição.

Em seguida, é possível executar o servidor de desenvolvimento do Django, utilizando o comando mostrado na Figura 36:

```
$ python manage.py runserver

Validating models...

0 errors found
Django version 1.3 rc 1, using settings 'teste.settings'
Development server is running at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

Figura 36 - Execução do servidor web do framework Django.

Fonte: PRÓPRIA (2012)

Por padrão o servidor *web* de desenvolvimento do Django atende as requisições na porta 8000. Caso seja necessário é possível atender requisições de outras máquinas além da que ele está rodando ou em outra porta, para isso, é necessário adicionar 0.0.0.0:[porta] ao comando *runserver*. Exemplo: `python manage.py runserver 0.0.0.0:8081`.

Em seguida, será verificado se o Django está funcionando corretamente, para isso é necessário acessar a *URL* `http://localhost:8000` a partir de um navegador *web*. Ao acessar esse endereço a página mostrada na Figura 37 deverá ser apresentada, indicando que o Django está funcionando corretamente.

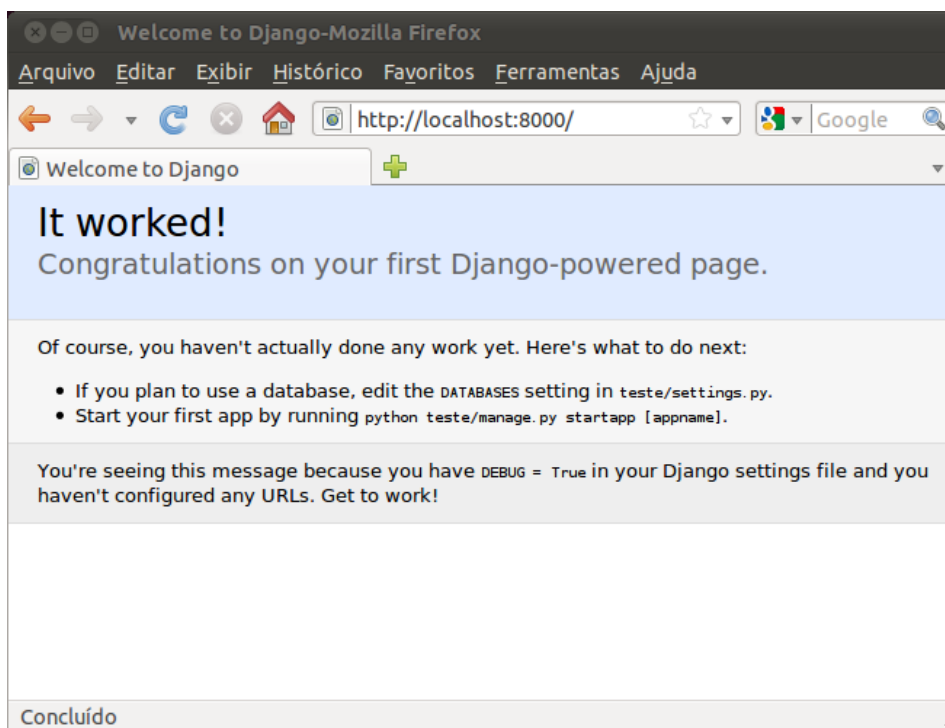


Figura 37 - Acesso ao servidor de desenvolvimento do Django

Fonte: PRÓPRIA (2012)

O servidor *web* do *framework* Django deve ser utilizado apenas para agilizar o desenvolvimento e não deve ser usado em ambientes de produção, pois se trata de um servidor simples e que não suporta grandes demandas de acesso. Alguns servidores que podem ser utilizados no ambiente de produção são: Apache, *lighttpd*, *FastCGI* e *nginx*.

### 3.3.2.1 APLICAÇÕES

O Django utiliza um conceito de projeto com várias aplicações. Isso significa que, dentro de cada projeto existem várias aplicações, que podem ser compreendidas como um componente desse projeto. Cada componente possui um escopo específico, com funcionalidades bem definidas, de forma que possam ser reutilizadas em vários outros projetos.

Os projetos desenvolvidos em Django tornam-se modularizadas devido à utilização de aplicações, que executam tarefas específicas do sistema. Por exemplo, no desenvolvimento de *sites* é costume utilizar uma página de contato para os usuários deixarem uma mensagem de dúvida ou sugestão para os administradores do *site*. Essa página de contato é utilizada em vários outros *sites*, podendo assim, se tornar uma aplicação.

Para criar uma aplicação é executado o seguinte comando, conforme mostrado na Figura 38:

```
$python manage.py startapp nomedaapp
```

Figura 38 - Comando para criar uma aplicação

Fonte: PRÓPRIA (2012)

Após executar o comando, é possível observar que foi criado um diretório com o nome da aplicação e neste diretório são criados os seguintes arquivos:

- `__init__.py` : arquivo vazio que indica ao Python que o diretório atual deve ser considerado como um pacote;
- `models.py`: arquivo utilizado para a definição dos modelos de dados utilizados pela aplicação;
- `views.py`: arquivo utilizado para a definição das *Views* da aplicação, que em conjunto com o *URL Dispatcher*, equivalem à camada *Controller* do modelo MVC.



Após criar a aplicação é necessário habilitá-la no projeto. Para isso, é necessário alterar o arquivo `settings.py`, adicionando a aplicação criada na lista de `INSTALLED_APPS`.

Na Figura 39 é apresentado o exemplo para adicionar uma aplicação ao projeto.

```
INSTALLED_APPS = (  
    #...  
    'meuprojeto.minhaapp',  
    #...  
)
```

Figura 39 - Configuração para adicionar uma aplicação ao projeto

Fonte: PRÓPRIA (2012)

### 3.3.3 CONFIGURANDO O PROJETO

Todas as configurações de um projeto desenvolvido com o Django são guardadas no arquivo `settings.py`. Algumas configurações já vem pré-configuradas com as opções mais comuns, porém, não com todas as configurações possíveis. No endereço <https://docs.djangoproject.com/en/1.3/ref/settings/> é possível encontrar a listagem de todas as configurações com suas respectivas funções.

O arquivo de configurações do Django não tem a necessidade de definir quaisquer configurações. Cada configuração possui um valor padrão. Esses valores padrões são armazenados em um arquivo global de configurações que fica armazenado no diretório de instalação do Django, abaixo da pasta `conf`. Esse arquivo é o `global_settings.py`.

Informações adicionais sobre as configurações podem ser encontradas no endereço <https://docs.djangoproject.com/en/1.3/topics/settings/>.

Para compilar as definições, o Django realiza os seguintes passos:

- Carregar as configurações padrões do arquivo `global_settings.py`.
- Carregar as configurações do arquivo `settings.py`, substituindo as definições globais pelas definições específicas do projeto.

A configuração primordial e mais básica a ser feita no projeto é o de acesso ao banco de dados. O Django suporta, nativamente, os bancos de dados: MySQL, PostgreSQL, SQLite3 e Oracle.

Para configurar o banco de dados é necessário modificar as seguintes configurações do `settings.py`:

- `DATABASE_ENGINE`: variável utilizada para informar qual o sistema de banco de dados utilizado. Os valores possíveis, por padrão, no Django são: `postgresql_psycopg2`, `postgresql`, `mysql`, `sqlite3` ou `oracle`;
- `DATABASE_NAME`: nome do banco de dados usado. Para o `sqlite3`, o caminho completo para o arquivo com o banco de dados;
- `DATABASE_USER`: nome do usuário para conexão com o banco de dados. Não é utilizado pelo `sqlite3`;
- `DATABASE_PASSWORD`: senha para conexão como banco de dados. Não é utilizado pelo `sqlite3`;
- `DATABASE_HOST`: nome do *host* ou IP do servidor de banco de dados. Não utilizado pelo `sqlite3`. Se esse valor não for informado, o Django assume que o banco de dados está sendo executado no mesmo host. E assumirá o valor *localhost* ou `127.0.0.1`;
- `DATABASE_PORT`: porta usada para conectar ao banco de dados. Se esse valor não for informado, o Django assume que o banco de dados será acessado pela porta padrão. Não é utilizado no `sq3lite`.

A Figura 40 mostra a configuração de um projeto para a utilização do banco de dados PostgreSQL.

```
DATABASES = {  
    'default': {  
        'ENGINE': 'django.db.backends.postgresql',  
        'NAME': 'exemplo',  
        'USER': 'postgres',  
        'PASSWORD': '123456',  
        'HOST': 'localhost',  
        'PORT': '5432',  
    }  
}
```

Figura 40 - Configuração para utilização do banco de dados PostgreSQL

Fonte: PRÓPRIA (2012)

### 3.3.4 MTV

Como já citado anteriormente, o Django trabalha com o paradigma MVC e nessa seção serão exemplificados a utilização de cada uma das três camadas do MVC correspondentes do Django.

#### 3.3.4.1 MODELS

Nessa seção será demonstrada a utilização dos modelos de dados do Django além de algumas informações relevantes sobre eles.

Na Figura 41 é mostrado o exemplo de definição do modelo Pessoa, que tem os campos nome e ultimo\_nome. Cada campo é definido como um atributo da classe, e cada atributo é mapeado para uma coluna do banco de dados.

```
from django.db import models

class Pessoa(models.Model):
    nome = models.CharField(max_length=30)
    ultimo_nome = models.CharField(max_length=30)
```

Figura 41 - Exemplo de definição de modelo

Fonte: PRÓPRIA (2012)

O modelo Pessoa acima criaria uma tabela conforme a SQL mostrado na Figura 42.

```
CREATE TABLE nomedaapp_pessoa (
    "id" serial NOT NULL PRIMARY KEY,
    "nome" varchar(30) NOT NULL,
    "ultimo_nome" varchar(30) NOT NULL
);
```

Figura 42 - SQL gerada a partir da definição do modelo

Fonte: PRÓPRIA (2012)

Algumas informações técnicas sobre os modelos do Django:

- O nome da tabela, nomedaapp\_pessoa, é automaticamente derivado de alguns metadados do modelo, para economizar o tempo do desenvolvedor, o Django

automaticamente deriva o nome das tabelas do banco de dados usando os nomes das classes *Models* e da aplicação que o contém, entretanto, isto pode ser sobrescrito, alterando o atributo `db_name` das metaopções da classe. Mais informações sobre o nome de tabelas podem ser encontradas no endereço <https://docs.djangoproject.com/en/1.3/ref/models/options/#table-names>.

- Um campo `id` é adicionado automaticamente a tabela, porém esse comportamento também pode ser alterado, basta adicionar o parâmetro `primary_key=True` ao campo escolhido para ser a chave primária. Mais informações sobre chaves primárias podem ser encontradas no endereço <https://docs.djangoproject.com/en/1.3/topics/db/models/#automatic-primary-key-fields>.
- O comando SQL `CREATE TABLE` nesse exemplo foi formatado usando a sintaxe do PostgreSQL, o Django usa o SQL adaptado ao banco de dados especificado no seu arquivo de configurações (`settings.py`).

Na Figura 43 são mostradas as opções disponíveis no modelo para adicionar chave primária e para escolher o nome da tabela que será criada no banco de dados.

```

1 from django.db import models
2
3 class Pessoa(models.Model):
4     nome = models.CharField(max_length=30)
5     ultimo_nome = models.CharField(max_length=30)
6     cpf = models.CharField(max_length=11, primary_key=True)
7
8     class Meta:
9         db_table = "tabela_pessoa"

```

Figura 43 - Opções disponíveis no model do Django

Fonte: PRÓPRIA (2012)

Após a definição do modelo, o passo seguinte é informar ao Django para usar estes modelos. Para isto, basta alterar o arquivo `settings.py` e mudar o `INSTALLED_APPS` adicionando o nome da aplicação que contém o `models.py`, conforme mostrado anteriormente na Figura 39.

Sempre após adicionar novas aplicações ao `INSTALLED_APPS`, deve-se executar o *script* `manage.py` com o comando `syncdb` a partir do diretório raiz do projeto, conforme a Figura 44, esse comando cria as tabelas no banco de dados com os respectivos campos dos modelos.

```
$ python manage.py syncdb
```

Figura 44 - Sincronização das tabelas no banco de dados de acordo com o modelo de dados  
Fonte: PRÓPRIA (2012)

A lista completa dos tipos de campos disponíveis no Django e dos seus respectivos argumentos podem ser encontrados no endereço <https://docs.djangoproject.com/en/1.3/ref/models/fields/#model-field-types>.

### 3.3.4.2 TEMPLATES

Nessa seção será mostrada a utilização dos *templates*, que são as interfaces de apresentação dos dados utilizados nos projetos Django.

Nos *templates* é possível a utilização de *Tags*, elas correspondem a “comandos” da linguagem de *template*, que possibilitam o desenvolvedor realizar algumas operações, tais como: criar texto de saída, fluxo de controle como iterações ou verificações. As *tags* são envolvidas pelos símbolos “{% %}”. O Django possui mais de 20 *tags* na linguagem de *templates*, a lista completa pode ser acessada pelo endereço <http://docs.djangoproject.org/topics/templates.html#tags>.

Além das *tags*, é possível utilizar variáveis que terão seus valores substituídos na geração do HTML, elas são envolvidas pelos símbolos “{{ }}”.

Uma *tag* que possui uma grande utilidade é a {% extends %}. Essa *tag* permite que um *template* herde de um *template* base. O sistema de *templates* do Django possui um mecanismo de herança similar ao de classes na programação orientado a objetos. A diferença entre elas é a que os *templates* não lidam com atributos e métodos, mas com “blocos” que podem ser sobrescritos pelo *templates* derivados (SANTANA; GALESI, 2010).

Na Figura 45 e Figura 46 são mostrados exemplos de um *template* base e de um *template* filho.

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html>
    <head>
        {% block cabecalho %}
            <title>Título da página</title>
        {% endblock %}
    </head>
    <body>
        {% block conteudo %}
            <!--conteúdo da página-->
        {% endblock %}
    </body>
</html>

```

Figura 45 - Exemplo de template pai

Fonte: PRÓPRIA (2012)

```

{% extends 'base.html' %}
{% block cabecalho %}
    <title> Lista de pessoas</title>
{% endblock %}
{% block conteudo %}
    {% for pessoa in pessoas %}
        <p>Nome: {{ pessoa.nome }} </p>
        <p>Sobrenome: {{ pessoa.ultimo_nome }}</p>
    {% empty %}
        <p>Não existem pessoas cadastradas</p>
    {% endfor %}
{% endblock %}

```

Figura 46 - Exemplo de template filho

Fonte: PRÓPRIA (2012)

### 3.3.4.3 VIEWS

Nessa seção serão mostradas as configurações necessárias para o funcionamento das *Views* que são correspondentes a camada de controle do MVC.

Na Figura 47 é mostrado o exemplo de uma *View* que retorna um HTML contendo a mensagem “*Hello World*”.

Exemplo:

```
1 from django.http import HttpResponse
2
3 def hello(request):
4     html = "<html><body>Hello World.</body></html>"
5     return HttpResponse(html)
```

Figura 47 - Exemplo de função View

Fonte: PRÓPRIA (2012)

Após definir a *View* é necessário configurar qual a *url* será responsável por chamá-la.

Para definir *URLs* para uma aplicação, é necessário criar um módulo chamado *URLConf*, um acrônimo para configuração de *URL*. Esse módulo é escrito em Python. Ele faz um simples mapeamento entre padrões de *URL* e as *Views* do projeto. Esse mapeamento pode referenciar outros mapeamentos e pode ser construído dinamicamente.

Para definir qual o módulo *URLConf* a ser usado, é necessário alterar o valor da configuração *ROOT\_URLCONF* do arquivo de configuração (*settings.py*), que, por padrão, é preenchido com o valor "nomedoprojeto.urls", correspondente ao arquivo *urls.py* localizado no diretório do projeto.

Na Figura 48 é mostrado o exemplo da configuração do arquivo *urls.py*.

```
1 from django.conf.urls.defaults import *
2
3 urlpatterns = patterns("",
4     #referência para outro mapeamento
5     (r'^minhaapp/', include('minhaapp.urls')),
6
7     #mapeamento da url inicial de um site/aplicacao
8     #url mapeada para a função index de uma view
9     (r'^$', 'home.views.index'),
10    (r'^$', 'minhaapp.views.hello'),#url mapeada para view hello
11 )
```

Figura 48 - Configuração do arquivo urls.py

Fonte: PRÓPRIA (2012)

### 3.3.5 ORM

O Django disponibiliza um *console* interativo para manipulação dos modelos juntamente com o banco de dados. Para poder utilizá-lo basta definir os modelos de dados que serão utilizados e sincronizá-los utilizando o *script* manage.py com o comando *syncdb*, que é executado para a criação das tabelas correspondente aos modelos. O *console* interativo permite executar ações para criar, alterar ou remover algum registro no banco de dados.

Para iniciar o console interativo do projeto, é necessário executar o *script* manage.py com a comando *shell* na raiz do projeto, conforme mostrado na Figura 49:

```
$ python manage.py shell
```

Figura 49 - Comando para iniciar o console interativo

Fonte: PRÓPRIA (2012)

Para exemplificar a utilização do *console* interativo, será criado um modelo referente à entidade Aluno. A Figura 50 mostra a definição do modelo de dados Aluno e a Figura 51 mostra a SQL utilizada para a criação da tabela desse modelo. A SQL foi gerada no padrão do banco de dados MySQL, de acordo com a configuração preenchida no arquivo settings.py.

```
from django.db import models

class Aluno(models.Model):
    nome = models.CharField(max_length=50)
    data_nascimento = models.DateField()
    matricula = models.CharField(max_length=5)
```

Figura 50 - Definição do modelo de dados Aluno

Fonte: PRÓPRIA (2012)

```
BEGIN;
CREATE TABLE 'app_aluno' (
  'id' integer AUTO_INCREMENT NOT NULL PRIMARY KEY,
  'nome' varchar(50) NOT NULL,
  'data_nascimento' date NOT NULL,
  'matricula' varchar(5) NOT NULL
)
;
COMMIT;
```

Figura 51 - SQL gerada pelo Django para criação da tabela de Aluno

Fonte: PRÓPRIA (2012)



Para gravar um registro no banco de dados através do Django, basta criar uma instância de uma classe que corresponde a tabela em que se deseja criar o registro, atribuir os valores necessários aos atributos do objeto e chamar o método *save()*. Os atributos do objeto são os campos definidos no modelo de dados. A Figura 52 mostra um exemplo de criação de um registro através do console interativo.

```
>>> from projeto.app.models import Aluno
>>> aluno = Aluno(nome='João',data_nascimento='2012-05-14',matricula='12345')
>>> print(aluno.nome)
João
>>> print(aluno.id)
0
>>> aluno.save()
>>> print(aluno.id)
1
```

Figura 52 - Criação de um registro na tabela Aluno

Fonte: PRÓPRIA (2012)

A atualização de registros possui um funcionamento similar ao de criação. A diferença está na atribuição de um valor para o campo da chave primária da tabela. Na execução do método *save()*, o Django verifica se o campo da chave primária está preenchido com algum valor e, caso esteja, faz uma busca no banco de dados de acordo com o valor informado. Se o registro for encontrado, este é atualizado. Caso não encontre, o Django cria o registro com os valores passados. Porém isso ocasiona um problema, como a consulta no banco de dados só acontece quando o Django faz chamada do método *save()*, é necessário informar os valores para todos os campos obrigatórios do registro, mesmo que seja necessário atualizar apenas um. Para resolver esse problema, é possível realizar a busca do registro, alterar o campo escolhido e salvar o objeto. Para buscar o registro utiliza-se o método *get()*.

Na Figura 53 é mostrado um exemplo que utiliza os dois métodos citados anteriormente para atualizar um registro no banco de dados.

```

>>> from projeto.app.models import Aluno
>>> aluno = Aluno(id=1,nome="Maria",data_nascimento="2000-05-16",matricula="54321")
>>> aluno.save()
>>> print(aluno.nome)
'Maria'
>>> print(aluno.id)
1
>>> aluno = Aluno.objects.get(id=1)
>>> print(aluno.nome)
'Maria'
>>> aluno.nome = 'Lucas'
>>> aluno.save()
>>> print(aluno.nome)
'Lucas'

```

Figura 53 - Atualização de um registro na tabela Aluno

Fonte: PRÓPRIA (2012)

Conforme mostrado na Figura 53, o método *get()* busca um registro específico no banco de dados. Porém, no ambiente de produção existe a necessidade de consultar todos os registros de uma tabela ou consultar registros de acordo com certa condição. A busca de registros no banco de dados é realizada pelo gerenciador do modelo de dados. Todo objeto do tipo *Model* possui um atributo chamado "*objects*", que aponta para uma instancia de *ModelManager*. A partir desse atributo é possível acessar todos os objetos de um determinado tipo e possibilita a criação de *queries* para recuperar dados. Esse objeto é utilizado para buscar um *QuerySet*, que representa uma coleção de vários objetos do banco de dados. O *QuerySet*, por sua vez, pode possuir filtros que irão delimitar a busca dos registros de acordo com as condições especificadas. Em uma analogia com a linguagem SQL, o *QuerySet* representa o comando *SELECT* e os filtros representam as cláusulas *WHERE* (SANTANA; GALESI, 2010).

A busca para retornar todos os registros de uma tabela pode ser feita utilizando o método *all()*, conforme o exemplo na Figura 54:

```
>>> from projeto.app.models import Aluno
>>> alunos = Aluno.objects.all()
>>> for aluno in alunos:
... print(aluno.nome)
...
'Maria'
'João'
'Lucas'
```

Figura 54 - Consulta a todos os registros da tabela Aluno

Fonte: PRÓPRIA (2012)

Durante o desenvolvimento de uma aplicação, é normal que o programador necessite acrescentar condições na consulta, para conseguir uma coleção de dados mais específica. O Django disponibiliza dois métodos para acrescentar estas condições ao *QuerySet*:

- *filter()*: faz *queries* que filtrem seus resultados, retorna um *QuerySet* com os registros que atendem as condições informadas nos parâmetros do método. Apesar de funcionar de maneira similar ao método *get()* o método *filter()* retorna um *QuerySet*.
- *exclude()*: funciona de maneira inversa ao método *filter()*, retornando um *QuerySet* com os registros que não atendem as condições informadas nos parâmetros do método.

A Figura 55 mostra exemplos da utilização dos métodos *filter()* e *exclude()*.

```
>>> from projeto.app.models import Aluno
>>> alunos = Aluno.objects.filter(nome='João')
>>> for aluno in alunos:
... print(aluno.nome)
...
'João'
>>> alunos = Aluno.objects.exclude(nome='João')
>>> for aluno in alunos:
... print(aluno.nome)
...
'Maria'
'Lucas'
'José'
```

Figura 55 - Consulta condicional aos registros da tabela Aluno

Fonte: PRÓPRIA (2012)

Até o momento todas as consultas realizadas foram filtradas por valores exatos. Para realizar consultas por valores parciais ou relativos, deve-se passar parâmetros especiais ao `filter()`. Esses parâmetros seguem o seguinte formato: `nome_do_campo__parâmetro`. O QUADRO 6 lista todas as condições de consulta disponíveis no Django.

QUADRO 6

Parâmetros especiais para consulta

(Continua)

Parâmetro	Descrição
<code>exact/ iexact</code>	Busca exata(sensível ao <i>case</i> ou não)( <code>nome__iexact="john cleese"</code> )
<code>gt/ lt</code>	Maior/menor que ( <code>idade__gt=18</code> )
<code>startswith/istartswith</code>	Começa com (sensível ao <i>case</i> ou não)( <code>nome__startswith="john"</code> )
<code>endswith/iendswith</code>	Termina com (sensível ao <i>case</i> ou não)( <code>nome__endswith="john"</code> )
<code>year/ month/ day</code>	Filtra campos de data pelo ano/mês/dia ( <code>data_pub__year=2010</code> )
<code>week_day</code>	Busca pelo dia da semana ( <code>data_pub__week_day=2</code> )
<code>contains/icontains</code>	“contém” o item informado ( <code>nome__icontains="john"</code> )
<code>in</code>	Valor está contido na sequência ( <code>id__in=[1,2,3]</code> )
<code>range</code>	Valor está em uma faixa ( <code>data_pub__range=(data_inicio, data_fim)</code> )
<code>isnull</code>	Valor é nulo ( <code>data_pub__isnull=True</code> )
<code>search</code>	Busca do tipo <i>full-text</i> ( <code>manchete__search="+Python -Monty"</code> )
<code>regex/iregex</code>	Busca por expressão regular ( <code>cmd=r"import.*Item"</code> )

Fonte: SANTANA E GALESI (2012)

Informações adicionais a respeito dos parâmetros especiais podem ser encontradas no endereço <https://docs.djangoproject.com/en/dev/topics/db/queries/>.

A exclusão de registros no Django é feita através do método `delete()`. Esse método pode ser invocado de um objeto relacionado a um registro do banco de dados ou pode ser um *QuerySet*. Quando o método `delete()` é chamado a partir de um *QuerySet*, todos os registros retornados pelo *QuerySet* serão apagados.

A exclusão de um registro utilizado como chave estrangeira em outros registros fará com que os registros que apontam para este objeto sejam excluídos junto com o ele. Este mecanismo do Django comporta-se da mesma forma que o *DELETE CASCADE* da linguagem

SQL. A partir da versão 1.3 esse comportamento cascata é personalizável através do argumento *on\_delete* ao *ForeignKey*.

### 3.3.6 INTERFACE DE ADMINISTRAÇÃO

Para utilizar a interface de administração do Django, o desenvolvedor precisa fazer algumas alterações no arquivo de configuração do projeto (*settings.py*). O primeiro passo é adicionar *django.contrib.admin* entre as aplicações instaladas na opção *INSTALLED\_APPS*, conforme mostrado na Figura 56.

```
INSTALLED_APPS = (  
    'django.contrib.auth',  
    'django.contrib.contenttypes',  
    'django.contrib.sessions',  
    'django.contrib.sites',  
    'django.contrib.messages',  
    'django.contrib.staticfiles',  
    'django.contrib.admin',  
)
```

Figura 56 - Configuração para habilitar a interface de administração

Fonte: PRÓPRIA (2012)

Após realizar estas modificações no arquivo de configuração, para adicionar a aplicação da área administrativa ao projeto, é necessário sincronizar o banco de dados para que as tabelas usadas por ela sejam criadas. Neste instante, o Django criará as tabelas necessárias e solicitará a criação de um usuário administrador para a interface de administração. A Figura 57 exibe o comando para sincronizar as tabelas com o banco de dados.

```
$ python manage.py syncdb
```

Figura 57 - Comando para sincronizar criar as tabelas da área administrativa

Fonte: PRÓPRIA (2012)

Por fim, é necessário alterar o arquivo *urls.py* para mapear a *URL* */admin/* para apontar para a interface de administração. Para isso, basta remover os comentários das linhas mostradas na Figura 58 do arquivo *urls.py* do projeto. A Figura 58 exibe as linhas que devem ser descomentadas para habilitar a interface de administração.

```
from django.contrib import admin
admin.autodiscover()

.
.
.
url(r'^admin/', include(admin.site.urls)),
.
.
.
```

Figura 58 - Configuração de URL para habilitar a interface de administração

Fonte: PRÓPRIA (2012)

Após executar todos os passos de configuração já é possível utilizar a interface de administração, basta iniciar (ou reiniciar) o servidor de desenvolvimento do Django, através do *script* `manage.py` com o comando `runserver` e acessar o endereço `http://localhost:8000/admin`. A tela de *login* da interface de administração deve ser exibida, conforme a Figura 59.

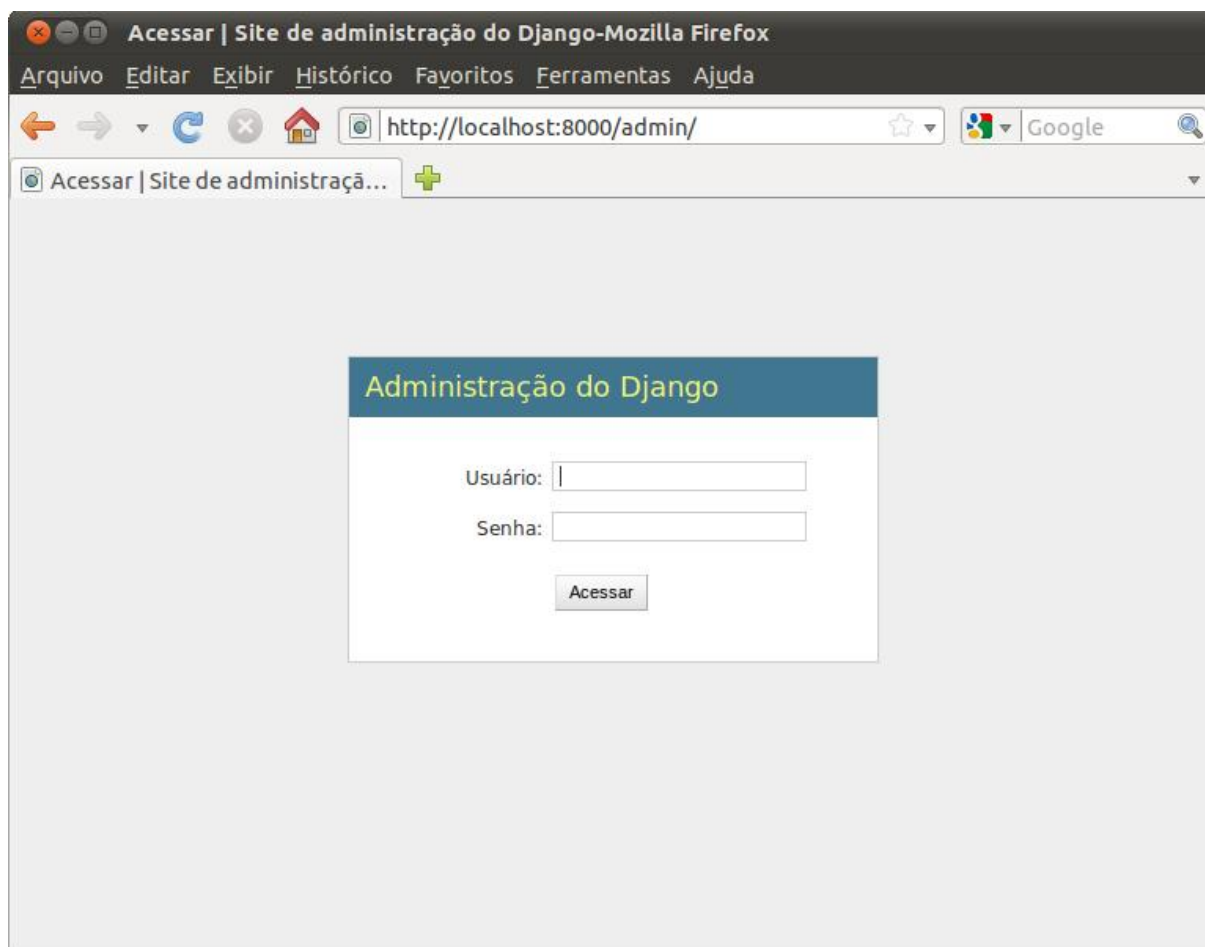


Figura 59 - Tela de login da interface de administração do Django

Fonte: PRÓPRIA (2012)

Para efetuar o *login*, é necessário utilizar o usuário e a senha criados na execução do *script* da Figura 57. Após entrar no sistema, a primeira tela da interface é a tela principal da área administrativa, mostrado na Figura 60.



Figura 60 - Tela principal da interface de administração do Django

Fonte: PRÓPRIA (2012)

As opções que aparecem na tela principal da interface de administração são referentes às tabelas padrões criadas pelo Django, que correspondem as aplicações `django.contrib.auth` e `django.contrib.sites` e aparecem no `INSTALLED_APPS` do arquivo de configuração.

Para adicionar suas próprias aplicações à tela principal é necessário criar uma aplicação, definir os modelos de dados da aplicação, registrá-la no `INSTALLED_APPS` e executar o *script* `manage.py` com o comando `syncdb`. Após executar todos esse procedimentos é necessário criar um arquivo chamado `admin.py` dentro do diretório da aplicação. O exemplo a seguir levará em conta que foi criado uma aplicação e que o modelo de dados foi o mesmo definido na seção 3.3.5 deste trabalho, que corresponde ao modelo de Aluno.

Após criar o arquivo `admin.py` dentro do diretório da aplicação o arquivo terá os seguintes códigos, apresentados na Figura 61.

```
1 from project.app.models import Aluno
2 from django.contrib import admin
3
4 admin.site.register(Aluno)
```

Figura 61 - Arquivo `admin.py`

Fonte: PRÓPRIA (2012)

Após criar o arquivo, é preciso recarregar a página da interface de administração. É possível perceber que foi criada mais uma entrada na tela principal correspondente a aplicação adicionada. Clicando no botão adicionar ao lado do *link* referente a aplicação criada, a página será direcionada para a tela de cadastro de um aluno, conforme a Figura 62.

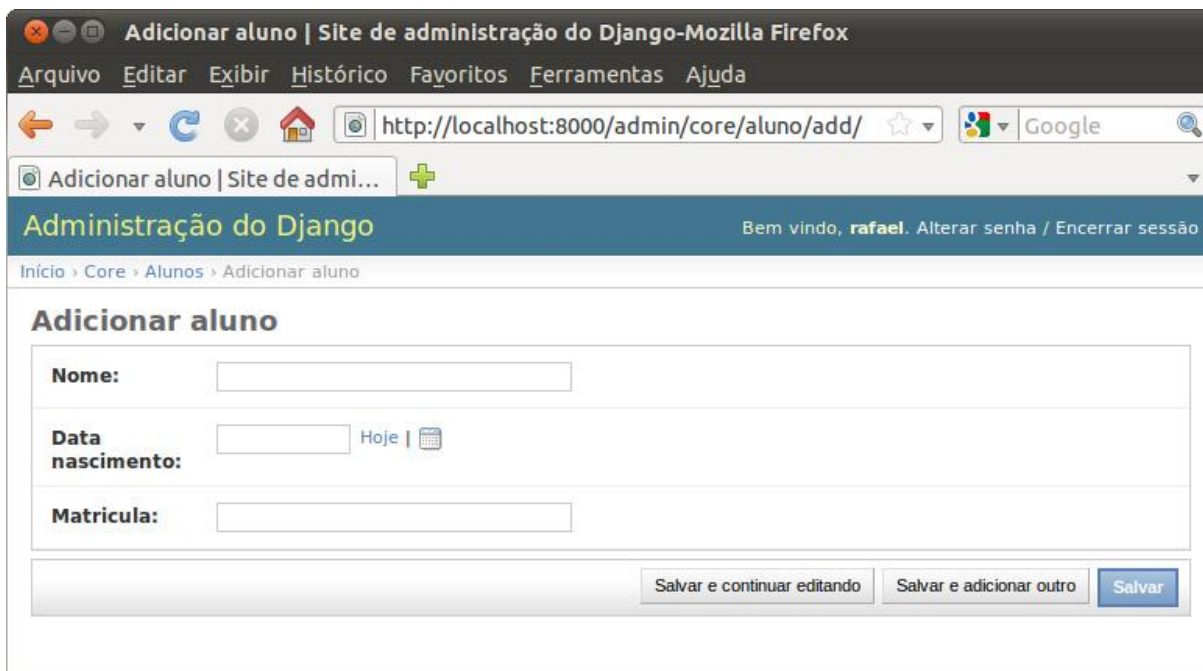
A imagem mostra uma interface web no navegador Mozilla Firefox. O título da aba é "Adicionar aluno | Site de administração do Django". O endereço da barra de endereços é "http://localhost:8000/admin/core/aluno/add/". O cabeçalho da interface exibe "Administração do Django" e o nome de usuário "Bem vindo, rafael." com links para "Alterar senha" e "Encerrar sessão". O breadcrumb indica o caminho: "Início > Core > Alunos > Adicionar aluno". O formulário principal, intitulado "Adicionar aluno", contém três campos de entrada: "Nome:" (um campo de texto simples), "Data nascimento:" (um campo de data com um ícone de calendário e o texto "Hoje" ao lado) e "Matricula:" (um campo de texto simples). Na base do formulário, há três botões: "Salvar e continuar editando", "Salvar e adicionar outro" e "Salvar".

Figura 62 - Formulário para cadastro de aluno

Fonte: PRÓPRIA (2012)

A interface criada para gerenciar os alunos já é bastante útil, permitindo o usuário adicionar, remover ou atualizar o registro dos alunos. No entanto, o Django ainda permite adicionar outros recursos às telas de cadastros e de listagem dos registros. Para isso, basta criar uma classe dentro do arquivo `admin.py` que herda da classe `ModelAdmin` e informar essa classe no momento de registrar a classe `Aluno`. Nesse arquivo, será definida a classe `AdminAluno`, que irá determinar quais os campos serão mostrados no formulário de cadastro e quais as colunas serão exibidas na listagem dos registros, conforme mostrado na Figura 63.



```
1 from project.app.models import Aluno
2 from django.contrib import admin
3
4 class AdminAluno(admin.ModelAdmin):
5     fields = ('nome','matricula','data_nascimento')
6     search_fields = ['nome','matricula']
7     list_filter = ['data_nascimento']
8     list_display = ('matricula','nome')
9
10 admin.site.register(Aluno, AdminAluno)
```

Figura 63 - Arquivo admin.py com opções adicionais

Fonte: PRÓPRIA (2012)

O atributo *fields* define quais os itens serão apresentados e qual a ordem de apresentação deles. O atributo *list\_display* defini quais campos serão exibidos na listagem de registros. O atributo *search\_fields* especifica quais campos farão parte da pesquisa na listagem de objetos, quando esse atributo não é definido, a pesquisa não aparece na listagem. O atributo *list\_filter* define quais serão os filtros que aparecerão na listagem dos registros, da mesma maneira que o *search\_fields*, se ele não for especificado, as opções de filtros não aparecerão na listagem dos registros.

Na Figura 64 é mostrado o resultado das alterações no arquivo admin.py. É possível ver as colunas na ordem e com os campos listados no atributo *list\_display*. Na parte superior é possível localizar a pesquisa, que foi adicionada para buscar os campos nome e matricula, de acordo com o campo *search\_fields*. E na barra lateral da direita foi adicionado o filtro por data de nascimento, de acordo com o atributo *list\_fields*.

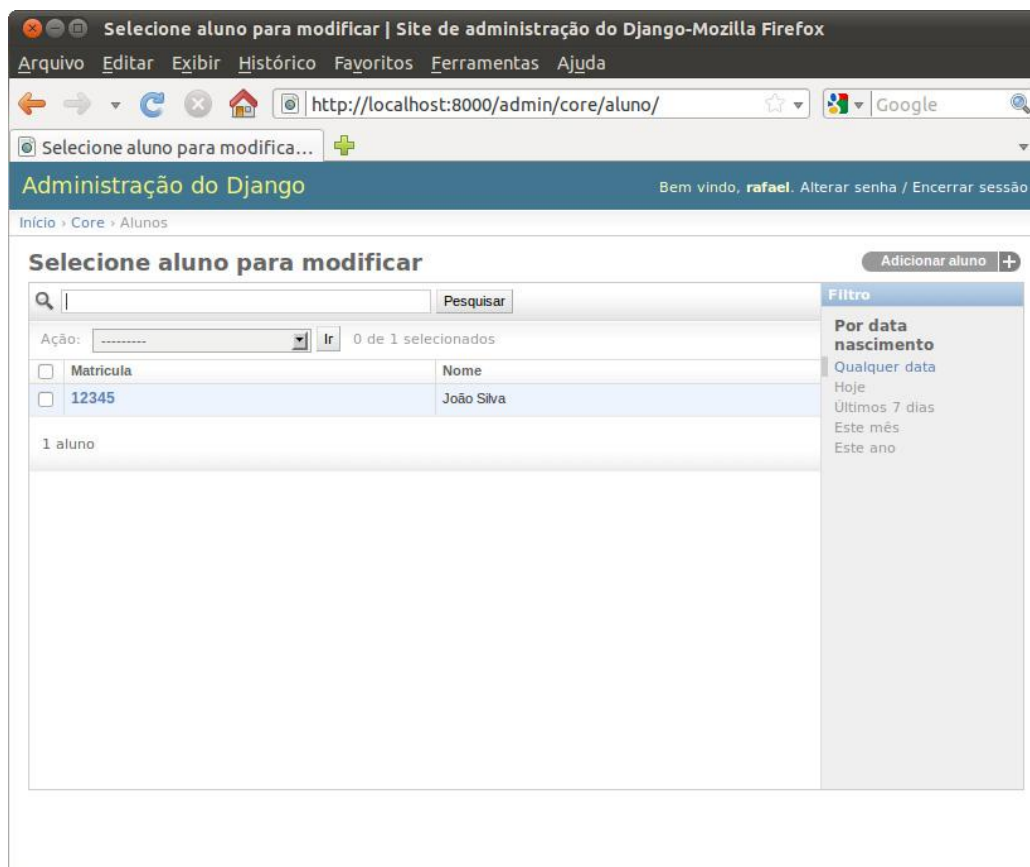


Figura 64 - Listagem dos registros com os recursos habilitados no arquivo admin.py  
Fonte: PRÓPRIA (2012)

Na Figura 65 é exibido o formulário de cadastro de aluno com os campos exibidos e ordenados de acordo com a sequência dos campos setados no atributo *fields*.

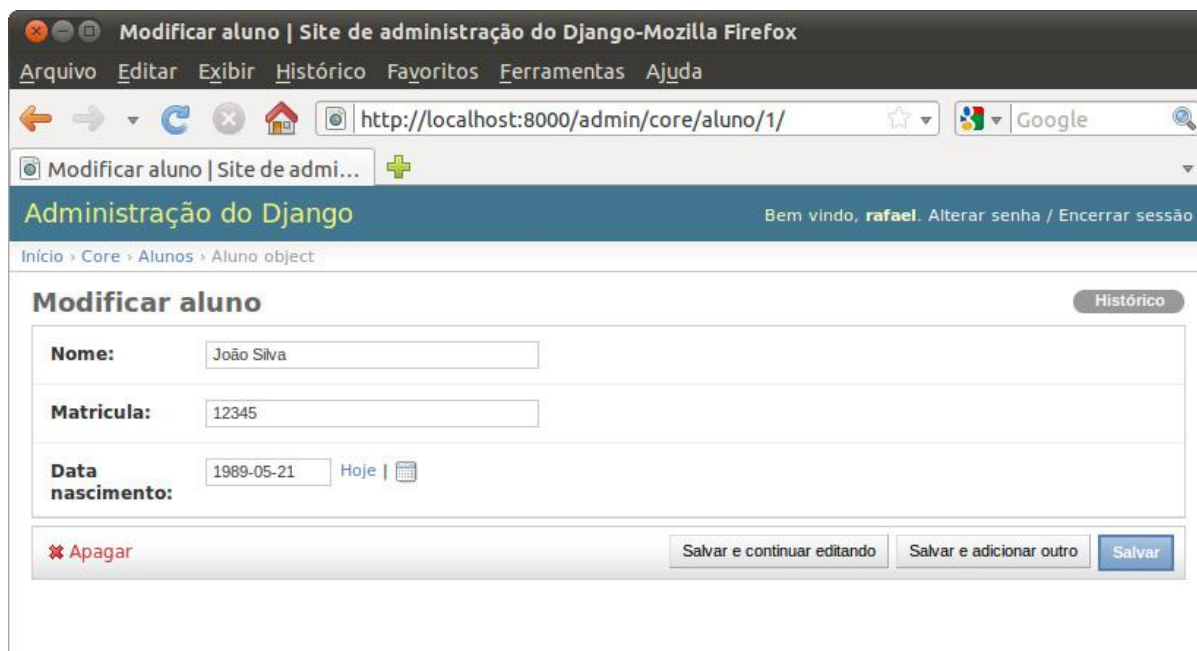


Figura 65 - Formulário de cadastro de aluno com os campos ordenados pelo atributo *fields*  
Fonte: PRÓPRIA (2012)

O Django possui várias outras configurações para a interface administrativa. A lista completa de opções está disponível no endereço <https://docs.djangoproject.com/en/1.3/ref/contrib/admin/>.

### 3.4 DESENVOLVIMENTO DA APLICAÇÃO DE DEMONSTRAÇÃO

Nessa seção serão mostrados os procedimentos básicos para desenvolver uma aplicação geográfica. Não há aqui a intenção de mostrar cada arquivo ou trecho de código da aplicação a ser desenvolvida, pretende-se dar uma breve descrição da arquitetura básica para a realização deste projeto.

O primeiro procedimento para desenvolver a aplicação de demonstração é criar um novo projeto. O nome do projeto será `mocmap`. Os comandos necessários para criar um projeto estão na seção 3.3.2.

O próximo procedimento é criar um banco de dados para o projeto. Conforme explicado anteriormente, o banco de dados que possui maior compatibilidade com o GeoDjango é o PostGIS, por isso, ele foi escolhido para ser o banco de dados da aplicação de demonstração. O banco de dados será criado com o mesmo nome do projeto. O novo banco de dados deve ser criado utilizando a base modelo criada pelo PostGIS no momento da instalação. Normalmente seu nome é `template_postgis`. A Figura 66 mostra a criação do banco de dados utilizando o modelo `template_postgis`.

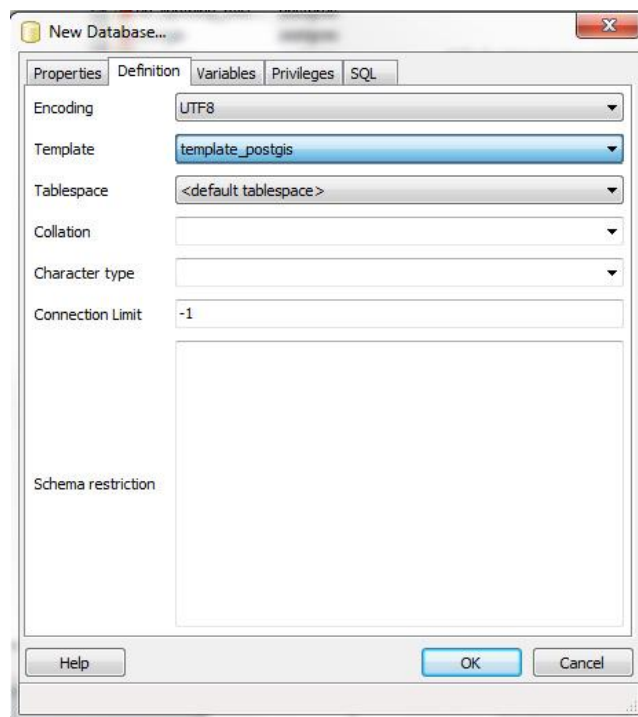


Figura 66 - Criação de uma base de dados utilizando o modelo do PostGIS.

Fonte: PRÓPRIA (2012)

Em seguida, serão adicionadas as configurações referentes ao banco de dados. Para utilizar o PostGIS é imprescindível a instalação de algumas bibliotecas, listadas no QUADRO 4. A Figura 67 exibe a configuração para a aplicação de demonstração.

```
DATABASES = {
    'default': {
        'ENGINE': 'django.contrib.gis.db.backends.postgis',
        'NAME': 'mocmap',
        'USER': 'postgres',
        'PASSWORD': '12345678',
        'HOST': 'localhost',
        'PORT': '5432',
    }
}
```

Figura 67 - Configuração do banco de dados da aplicação de demonstração

Fonte: PRÓPRIA (2012)

Em seguida, para verificar se o projeto foi configurado corretamente, é necessário acessar o endereço <http://localhost:8000>. Se o Django estiver funcionando corretamente, a tela apresentada será a mostrada na Figura 37.

Logo após criar o projeto, será criada a aplicação que ficará responsável pelas funcionalidades do mapa. O nome da aplicação será *core*. Como a aplicação de demonstração

é relativamente simples, não existe necessidade de criar outra aplicação para dividir as funcionalidades. Os passos necessários para criar uma aplicação são explicados na seção 3.3.2.1.

Para o funcionamento do GeoDjango é preciso adicionar ao `INSTALLED_APPS` do arquivo de configuração(*settings.py*) a aplicação *admin.contrib.gis*. Após a adição dessa entrada na lista de aplicações, será executado o *script* `manage.py` com o comando *syncdb* para a sincronização das tabelas.

Em seguida, será definido o modelo de dados. A Figura 68 mostra o modelo de dados definido para a aplicação de demonstração.

```

1  from django.contrib.gis.db import models
2  class Pontos(models.Model):
3      titulo = models.CharField(verbose_name=u'Título', max_length=80, blank=False)
4      descricao = models.CharField(verbose_name=u'Descrição', max_length=150, blank=False)
5      geometria = models.GeometryField(verbose_name=u'Ponto', null=False)
6      visivel = models.BooleanField(verbose_name=u'Visível', default=True)
7      atualizado_em = models.DateTimeField(auto_now=True, verbose_name=u'Atualizado')
8      link = models.CharField(verbose_name=u'Url de video do Youtube,max_length=200,blank=True)
9      imagem = models.ImageField(upload_to = 'upload/imagens',blank=True)
10     objects = models.GeoManager()
11
12     class Meta:
13         verbose_name = 'Ponto'
14         verbose_name_plural = 'Pontos'
15     def __unicode__(self):
16         return self.title

```

Figura 68 - Modelo de dados da aplicação de demonstração

Fonte: PRÓPRIA (2012)

Em seguida, é necessário executar o *script* `manage.py` com o comando *syncdb* para criar as tabelas do modelo no banco.

Uma diferença da declaração desse modelo em comparação ao demonstrado na seção 3.3.4.1 é que esse modelo herda as propriedades do módulo GIS, onde são adicionados os tipos geográficos.

O próximo procedimento é a ativação da área administrativa. Os procedimentos para ativar a área administrativa são explicados na seção 3.3.6. Em seguida, será criada a área para gerenciar os dados da aplicação *core*. Para isso, será criado o arquivo `admin.py` dentro do diretório da aplicação. O código do arquivo `admin.py` é mostrado na Figura 69.

```

1  from django.contrib import admin
2  from core.models import Pontos
3
4  class PontosAdmin(admin.ModelAdmin):
5      list_filter = ['visivel ', 'atualizado']
6      list_display = ('titulo ', 'atualizado', 'geometria', 'visivel')
7
8  admin.site.register(Pontos, PontosAdmin)

```

Figura 69 - Arquivo admin.py da aplicação de demonstração

Fonte: PRÓPRIA (2012)

Prosseguindo com o desenvolvimento da aplicação de demonstração, serão criadas 3 funções no arquivo *views.py* da aplicação *core*. As funções serão: *index*, *view*, *new*. A função *index* será responsável pela página inicial da aplicação, a sua função é buscar todos os pontos gravados no banco de dados e mandá-los para o *template* de exibição, que contém um mapa. A função *new* é responsável pela operação de salvar um ponto escolhido no mapa e a função *view* é responsável por recuperar a informação de um ponto escolhido no mapa. Serão criados 3 *templates* correspondentes a cada uma das funções da *views*. No *template* da função *index* será implementado um mapa onde serão plotados os pontos retornados no banco de dados. No *template* da função *new* será implementado um formulário de cadastro e no *template* da função *view* será implementado um formulário para visualização das informações do ponto retornadas do banco.

Após definir as funções das *views*, é necessário mapear as suas *URLs*. Na Figura 70 é mostrado o mapeamento das *urls* para as *views* criadas.

```

1  from django.conf.urls.defaults import patterns, include, url
2
3  from django.contrib import admin
4  admin.autodiscover()
5
6  urlpatterns = patterns("",
7      url(r'^$', 'core.views.index', name='index'),
8      url(r'^new/', 'core.views.new', name="new"),
9      url(r'^view/(?P<ponto_id>\d+)/$', 'core.views.view'),
10     url(r'^admin/', include(admin.site.urls)),
11 )

```

Figura 70 - Configuração do arquivo urls.py da aplicação de demonstração

Fonte: PRÓPRIA (2012)

## 4 RESULTADOS OBTIDOS

Este trabalho permitiu a realização de um estudo sobre o *framework* Django e suas principais características para o desenvolvimento *web* e para o desenvolvimento de aplicações geográficas.

Através deste trabalho, também foi possível a construção de um protótipo de aplicação geográfica em um período de 15 dias, após um período de 3 meses de estudo sobre as tecnologias, ferramentas e bibliotecas utilizadas no desenvolvimento.

O objetivo do trabalho foi alcançado com sucesso e demonstrou a viabilidade de se criar aplicações geográficas utilizando o Django, GeoDjango, PostgreSQL com o PostGIS e a API do Google Maps.

A aplicação desenvolvida teve o propósito de mostrar a integração entre as tecnologias propostas, bem com, mostrar o potencial do *framework* Django no desenvolvimento *web*.

A solução implementada poderá ser aprimorada com a possibilidade de agregar mais funcionalidades e recursos, tais como: relatórios, gráficos e marcações de geometrias mais complexas. De acordo com a demanda específica de cada sistema.

Ao acessar a página principal da aplicação, serão carregados todos os pontos, conforme mostrada na Figura 71.

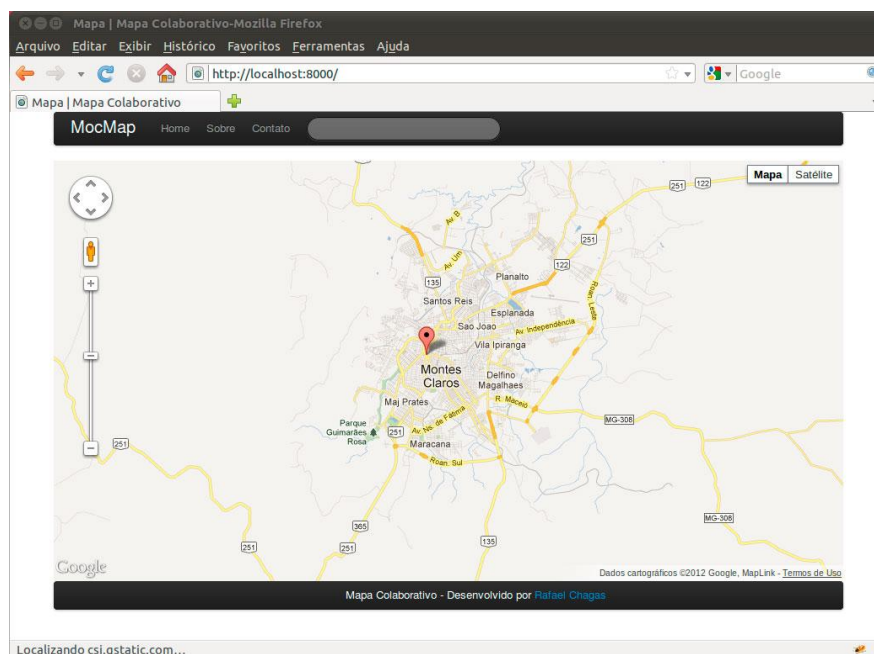


Figura 71 - Página principal do sistema

Fonte: PRÓPRIA (2012)

Para adicionar um novo ponto no mapa o usuário deve dar duplo clique no local desejado no mapa, assim, o usuário será redirecionado para a página de inclusão das informações sobre o ponto. Na Figura 72 é mostrada a página para a inclusão de um ponto com suas respectivas informações. Após finalizar a inclusão do ponto o usuário será redirecionado para a página principal.

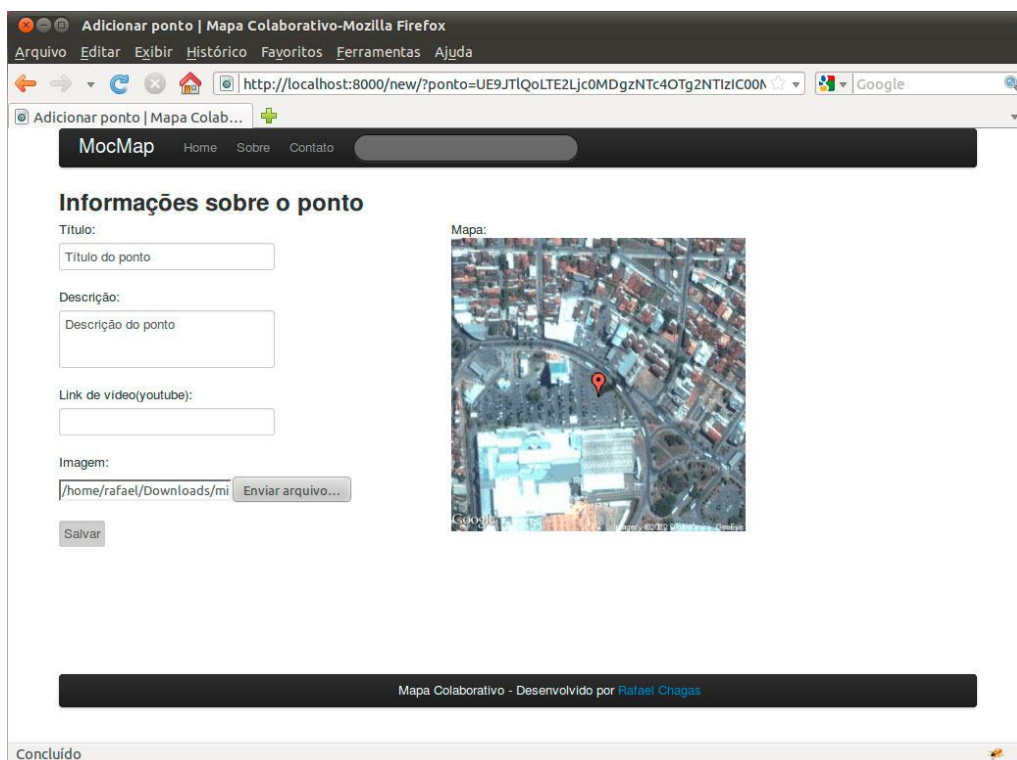


Figura 72 - Página para inclusão de um ponto

Fonte: PRÓPRIA (2012)

Para visualizar as informações sobre um ponto, o usuário deve clicar sobre um ponto desejado e em seguida clicar no botão Ver Detalhes, conforme mostrado na Figura 73.



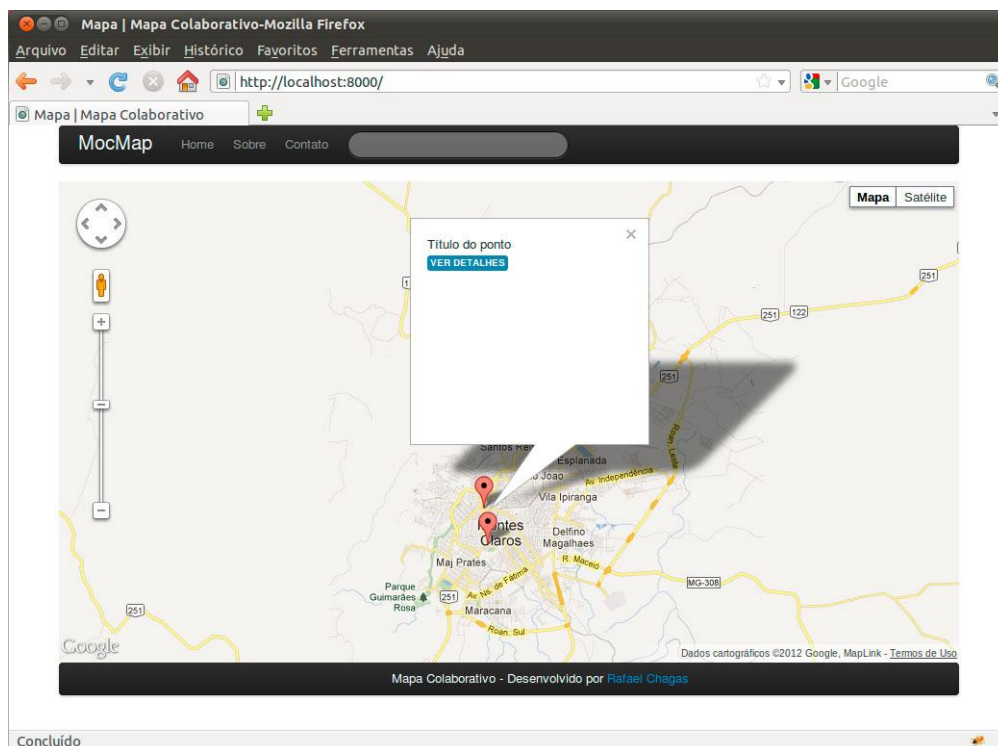


Figura 73 - Ponto selecionado no mapa

Fonte: PRÓPRIA (2012)

Após selecionar o ponto desejado, o usuário será redirecionado para uma página que exibirá todas as informações sobre o ponto, conforme mostrado na Figura 74.

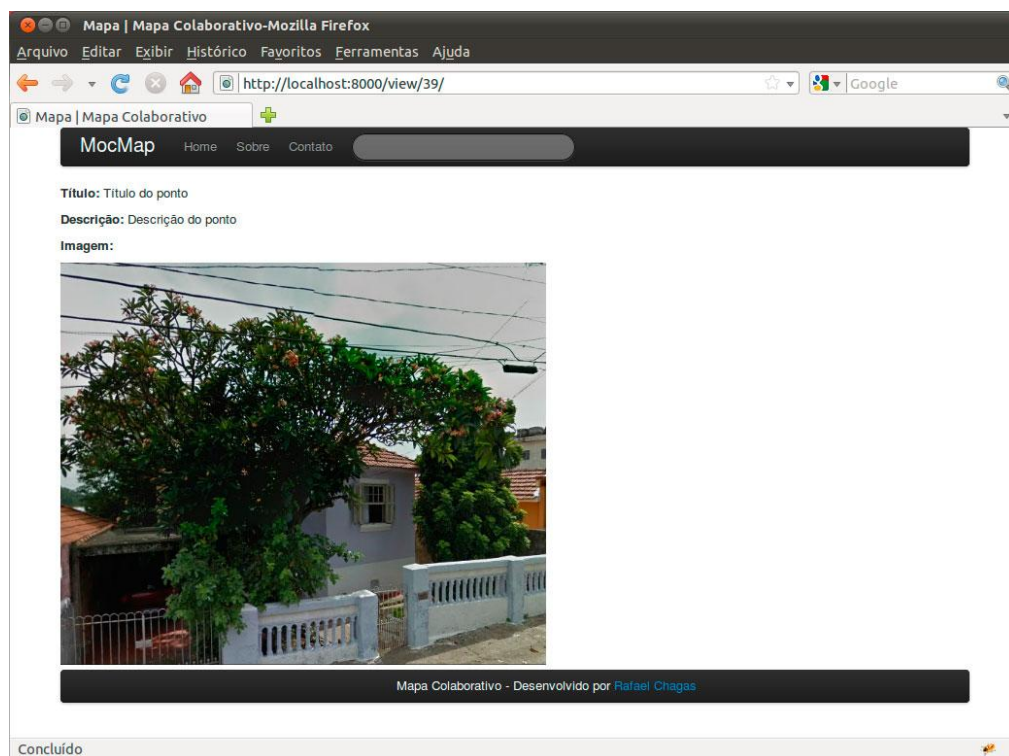


Figura 74 - Visualização das informações de um ponto

Fonte: PRÓPRIA (2012)

A interface da área administrativa da aplicação foi gerada automaticamente pelo *framework* Django, conforme as especificações do arquivo `admin.py`, mostradas na Figura 75 e Figura 76.

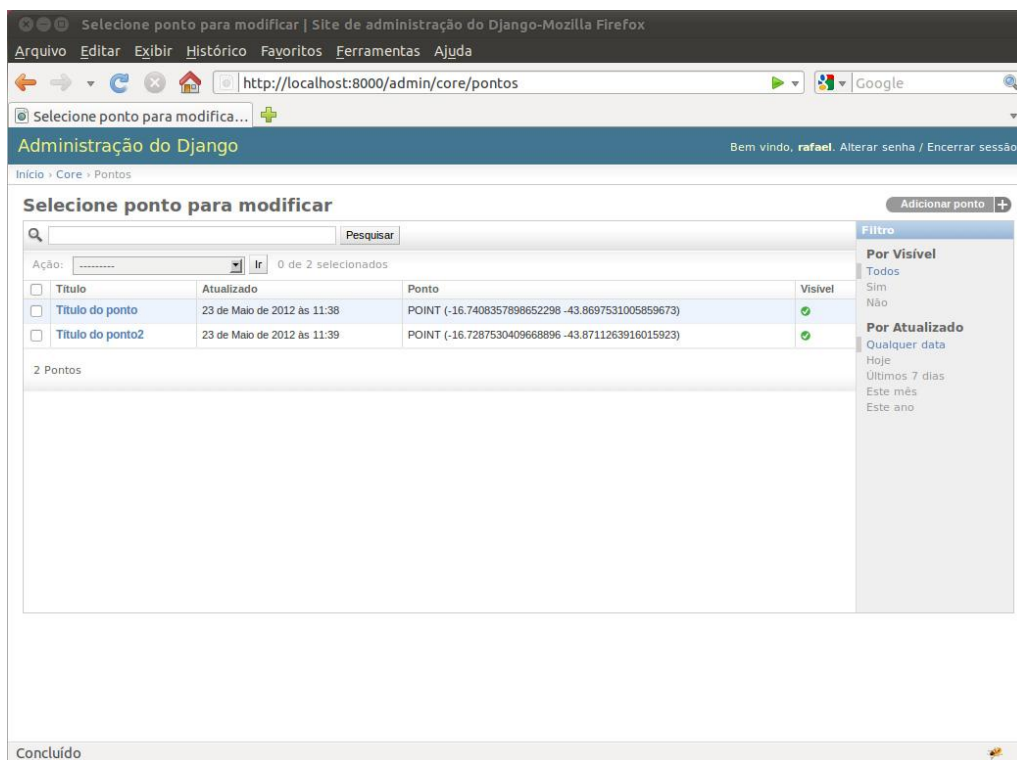


Figura 75 - Tela de listagem dos registros

Fonte: PRÓPRIA (2012)

Figura 76 - Formulário de cadastro/edição

Fonte: PRÓPRIA (2012)

## 5 CONSIDERAÇÕES FINAIS

Este projeto foi realizado com o intuito de se fazer um estudo sobre o *framework* Django e sua utilização no desenvolvimento de aplicativos geográficos, além do desenvolvimento de um protótipo de aplicação geográfica. Com isto, foi possível visualizar na prática as vantagens oferecidas pelo Django, tais como: geração da interface administrativa, acesso ao banco de dados utilizando o ORM, mapeador de *URLs*, redução do tempo de desenvolvimento, entre outras.

O módulo GeoDjango permitiu estender o modelo de dados do Django, adicionando o suporte a objetos geográficos, além de adicionar extensões para o ORM do Django que permitiu consultar e manipular dados espaciais. O PostgreSQL, com sua extensão espacial PostGIS, mostrou-se compatível para trabalhar com o GeoDjango. A API do Google Maps permitiu a visualização dos pontos, mostrando-se uma poderosa ferramenta para a interação com mapas.

A utilização de um *framework* baseado em convenções permitiu alcançar um alto nível de produtividade na criação das páginas da aplicação. Permitiu também oferecer um provedor de persistência com capacidade de fazer o mapeamento dos modelos de dados para o modelo relacional do banco de dados, que resultou em uma economia considerável de tempo e esforço necessários para a construção da camada de persistência, além de evitar a manipulação direta do banco de dados, que possibilitou uma maior independência da aplicação com o servidor de banco de dados, permitindo ainda, que a aplicação possa ser facilmente migrada para outros sistemas de banco de dados.

Dentre as principais dificuldades encontradas durante a realização deste trabalho, encontra-se a escassez de material escrito em língua portuguesa.

Para trabalhos futuros, sugere-se o desenvolvimento de uma aplicação robusta utilizando altos requisitos de desempenho, atestando o tempo das consultas do ORM do Django e propondo soluções de melhorias para as consultas que não atendem aos requisitos propostos, podendo ser utilizado o recurso de consultas manuais do Django para isso. Outra sugestão é o desenvolvimento de uma aplicação para dispositivo móvel que utilize as mesmas interfaces de comunicação da versão *web* normal, mostrando uma das grandes vantagens de se utilizar o paradigma MVC, permitindo que seja trocado a interface de exibição dos dados sem prejuízos a implementação do servidor.

## REFERÊNCIAS

AZEVEDO, C. **Google Maps API**, 2008. Disponível em: <<http://imasters.com.br/artigo/7832/linguagens/google-maps-api>>. Acesso em: 20 de maio de 2012.

BORGES, Luiz. **Python para desenvolvedores**. Rio de Janeiro, 2009.

BRONN, Justin. **GeoDjango: Web Applications for Geographers with Deadlines**. Slides de apresentação. Texas GIS Forum. Outubro, 2008. Disponível em: <<http://geodjango.org/presentations/>>. Acesso em 22 de maio de 2012.

BROWN, D. et al. **Struts 2 in Action**. 2008.

BUDD, Timothy A. **Exploring Python**. 1. ed. McGraw-Hill. 2009.

CÂMARA, et al. **Banco de Dados Geográficos**. 1.ed. Curitiba: MundoGeo, 2005.

CÂMARA, G. et al. **Anatomia de sistemas de informação geográfica**. Rio de Janeiro: SBC, 1996. Disponível em: <<http://www.dpi.inpe.br/gilberto/livro/anatomia.pdf>>. Acesso em: 14 de abr. de 2012.

CATUNDA, Marco. **Guia de consulta rápida Python**. São Paulo: Novatec, 2001.

DARIE, Cristian. et al. **AJAX and PHP: Building Responsive Web Applications**. Birmingham: Packt Publishing, 2006.

DEITEL, H. et al. **Python How To Program**. New Jersey: Prentice Hall, 2002.

DEITEL, Paul; DEITEL, Harvey. **Ajax, Rich Internet Applications and Web development for programmers**. Editora Pearson Prentice Hall, 2008.

**DJANGOPROJECT**. Disponível em: <<http://docs.djangoproject.com/>>. Acesso em: 28 mar. 2012.

ELMASRI, Ramez; NAVATHE, Shamkant B. **Sistemas de Banco de Dados**. 6.ed. São Paulo: Pearson Addison Wesley, 2011.

FAYAD, Mohamed E.; SCHMIDT, Douglas C.; JOHNSON, Ralph E.; **Building Application Frameworks: Object-Oriented Foundations of Framework Design**. 1999.

FOWLER, Martin. *The new methodology*. Disponível em: <<http://www.martinfowler.com/articles/newMethodology.html>>. Acesso em: 23 de maio de 2012.

GAMMA, Erich. **Padrões de projeto: soluções reutilizáveis de software orientado a objetos**. Porto Alegre: Bookman, 2000.

GOOGLE DEVELOPERS. **Google Maps JavaScript API V3**. Disponível em: <<http://code.google.com/intl/pt-BR/apis/maps/documentation/javascript/tutorial.html>>. Acesso em: 13 de maio de 2012.

GONÇALVES, Edson. **Desenvolvendo Aplicações Web com JPS, Servlets, JavaServer Faces, Hibernate EJB 3 Persistence e Ajax**. Rio de Janeiro: Ed. Ciência Moderna, 2007.

HOLOVATY, A. ; MOSS, J. K. . **The Django Book**. 2006. <Disponível em: <http://www.djangobook.com/>>. Acesso em: 20 de nov. de 2011.

HOLOVATY, A. ; MOSS, J. K. **The Definitive Guide to Django: Web Development Done Right**. 2.ed. 2009.

LUTZ, Mark; ASCHER, David. **Aprendendo Python**. 2. ed. Porto Alegre: Editora Bookman, 2007.

MOORE, D.; BUDD, R.; WRIGHT, W. **Professional Python frameworks : Web 2.0 programming with Django and Turbogears**. 1. ed. Editora Wrox, 2007.

OBE, Regina O. ; HSU, Leo S. **PostGIS in Action**. Manning Publications, 2011.

OGC. **OpenGIS Consortium Inc**. Disponível em: <<http://www.opengeospatial.org>>. Acesso em: 10 de maio de 2012.

PICK, James B. **Geographic Information Systems in Business**. Hershey, PA, USA. University of Redlands, 2005.

**POSTGIS.** Documentação oficial. Disponível em: <<http://postgis.refractory.net/documentation/>>. Acesso em: 18 de maio de 2012.

**PYTHON.** Documentação oficial. Disponível em: <<http://www.python.org>>. Acesso em: 28 de mar. de 2012.

SANTANA, Osvaldo; GALESI, Thiago. **Python e Django**. São Paulo: Novatec Editora. 2010.

SAUVÉ, J. P. **Frameworks**. Disponível em: <<http://www.dsc.ufcg.edu.br/~jacques/cursos/map/html/frame/oque.htm>>. Acesso em: 16 de maio de 2012.

TANENBAUM, Andrew; STEEN, Maarte Van. **Sistemas Distribuídos: princípios e paradigmas**. 2. ed. Editora Pearson Prentice Hall, 2007.