UNIVERSIDADE ESTADUAL DO SUDOESTE DA BAHIA
DEPARTAMENTO DE CIÊNCIAS EXATAS E TECNOLÓGICAS
CURSO DE CIÊNCIA DA COMPUTAÇÃO

DINO DOUGLAS RODRIGUES DE AGUILAR

DESENVOLVIMENTO DE UM SAAS NA NUVEM POR MEIO DA INTEGRAÇÃO DO GOOGLE APPENGINE COM O FRAMEWORK DJANGO

Vitória da Conquista – BA
2013

DINO DOUGLAS RODRIGUES DE AGUILAR

DESENVOLVIMENTO DE UM SAAS NA NUVEM POR MEIO DA INTEGRAÇÃO DO GOOGLE APPENGINE COM O FRAMEWORK DJANGO

Monografia de graduação apresentada ao Colegiado de Ciência da Computação da Universidade Estadual do Sudoeste da Bahia, como parte das exigências do curso para obtenção do título de Bacharel em Ciência da Computação.
Orientador: Roque Mendes Prado Trindade

Vitória da Conquista – BA
2013

RESUMO

Esta pesquisa tem como objetivo mostrar diferentes métodos de integração entre a plataforma de desenvolvimento na nuvem(PaaS) *Google App Engine* e o *framework Django*.Identificando as especificidades e os métodos encontrados para tornar possível o uso de um framework MVC de desenvolvimento rápido em uma plataforma que provê ao desenvolvedor todas as vantagens dos recursos da nuvem (escalabilidade, confiabilidade, recursos ilimitados, acesso global, agilidade).Para testar a integração foi realizada a conversão de um sistema escrito para a plataforma do *Google App Engine* para o *Django*. Utilizando-se o *fork Django-nonrel* que se apresentou entre todos o mais completo.

Palavras-Chaves: Computação em nuvem. *Google App Engine. Django. Python*

ABSTRACT

This research aims to show different methods of integration between the development platform in the cloud (PaaS) and *Google App Engine Django* framework. Identifying the methods found to make it possible to use an MVC framework for rapid development on a platform that provides the developer all the advantages offered by cloud services (scalability, reliability, limitless resources, global access, agility).To test the integration was carried out the conversion of a system developed for the platform *Google App Engine* to a system *Django*. Using the fork *Django*-nonrel who presented the most complete among all.
Keywords: Cloud computing, *Google App Engine*, *Django*, *Python*

SUMÁRIO

1. INTRODUÇÃO

O termo *Cloud Computing* (Computação em nuvem) foi criado em 2006 por Erick Schimidt (CEO da *Google*), enquanto falava em uma palestra sobre como a empresa gerenciava seus *data centers.*

Segundo *HAYES (2009)* o termo *Cloud Computing* tem se tornado popular e está associado à utilização da rede mundial de computadores com uso massivo de servidores físicos ou virtuais – uma nuvem – para a alocação de um ambiente de computação.

Hoje em dia a Computação em nuvem se encontra no centro de um movimento de importantes mudanças no mundo da tecnologia, pois com ela a *Internet* converge para se torna um agente centralizador, onde as pessoas poderão usar para armazenar seus arquivos, criar seus documentos, compartilhar suas fotos ou até mesmo usar um sistema operacional na nuvem sem precisar instalar qualquer *software* em seu computador.

O termo *Cloud Computing* está associado a outros conceitos como: *Software* como um Serviço (SaaS), Plataforma como um Serviço (PaaS) e Infra-Estrutura como um Serviço (IaaS), entre os quais será destacado o primeiro.

SaaS é um *software* que fornece serviços para o usuário final e é disponibilizado por servidores em *data centers* que ficam sob responsabilidade de uma empresa desenvolvedora, contudo, o *software* é desenvolvido por uma empresa que ao invés de comercializá-lo para um pequeno número de usuários com um alto custo, disponibiliza-o a um baixo custo a um grande número de usuários.

Um SaaS geralmente é executado em uma plataforma (PaaS) de desenvolvimento que facilita a implantação e o gerenciamento do *hardware* e das camadas de *software* e que fornece para o desenvolvedor condições necessárias para suportar o ciclo de desenvolvimento da aplicação, disponibilizando a ele uma *API*[1] que fornece uma camada de abstração de acesso aos recursos de *hardware* e *software* através de interfaces bem

[1] **API** – Application Programming Interface. Interface para desenvolvimento de aplicações.

definidas e documentadas e ferramentas que facilitem a criação e teste de *software*.

Atualmente existem muitas soluções de PaaS no mercado: *Google App Engine, Microsoft Azure, Amazon EC2, Salesforce Cloud, Engine Yard* entre outras, cada uma delas suporta um conjunto de plataformas e linguagens de programação, o que facilita a migração de sistemas legados e/ou a criação de novos sistemas sem que seja necessário fornecer treinamento para os funcionários.

Para resolver o problema formulado nessa pesquisa foi escolhido o *PaaS Google App Engine* que atualmente suporta as linguagens de programação *Python* versão 2.7 e Java versão 1.6 e a linguagem Go de forma experimental. Entre as quais foi escolhida *Python* por ser uma linguagem de programação de alto nível, orientada a objetos, interpretada, imperativa, de tipagem dinâmica e forte que combina sintaxe concisa e clara, com poderosos recursos providos pela sua biblioteca padrão e por *frameworks* e módulos desenvolvidos por terceiros.

O *Google App Engine* fornece para o desenvolvedor um interpretador *Python* completo, porém restringe o acesso às chamadas de sistemas e para acessar o banco de dados é usada uma linguagem própria que se assemelha ao SQL[2] chamada GQL (*Google Query Language*). Estes dois fatores dificultam a integração da plataforma com *frameworks* (*Django, web2py*) que já são consolidados entre os desenvolvedores *Python*.

Este trabalho propõe o desenvolvimento de um *SaaS* usando a infra-estrutura do *appspot.com* e a plataforma *Google App Engine Python* promovendo a integração com o *Django* com o objetivo de facilitar e agilizar o desenvolvimento do sistema usando um *framework* que oferece suporte ao padrão MVC[3] (*model-view-controller*).

[2]**SQL:** *Structured Query Language*, ou Linguagem de Consulta Estruturada, é uma linguagem de pesquisa declarativa para banco de dados relacional (base de dados relacional).

[3] **MVC:** È um modelo de desenvolvimento de Software. O modelo isola a "lógica" (A lógica da aplicação) da interface do usuário (Inserir e exibir dados), permitindo desenvolver, editar e testar separadamente cada parte

1.1 OBJETIVOS

1.1.1 OBJETIVO GERAL

Desenvolver um sistema na infraestrutura do *Google* usando a plataforma do *Google App Engine*.

1.1.2 OBJETIVOS ESPECÍFICOS

- Aplicar os Conceitos do modelo *SaaS*;
- Identificar as especificidades do *Google App Engin;*.
- Promover a integração da plataforma com outro *Framework* (*Django*).

1.2 JUSTIFICATIVA

A computação em nuvem é uma área nova e vem se tornando uma tendência da computação, pois promove a democratização de serviços e oferece aos desenvolvedores uma ferramenta com as vantagens de sistemas distribuídos (escalabilidade, disponibilidade, transparência, etc) e com uma grande variedade de opções de plataformas (*Google App Engine, Microsoft Azure, Amazon EC2, Engine Yard*, etc).

As plataformas de desenvolvimento na nuvem permitem que equipes desenvolvam e testem o *software* de maneira colaborativa e permite que pequenos desenvolvedores tenham acesso a recursos em uma infraestrutura com grande poder computacional.

1.3 METODOLOGIA

Esse trabalho foi desenvolvido usando o método indutivo com pesquisa bibliográfica e estudo de caso.

Foram realizadas pesquisas bibliográficas em livros, artigos e *sites* da *internet*. Posteriormente foi realizado o projeto de integração do *framework web Django* com a infraestrutura do *Google App Engine*.

Na terceira etapa foi modelado um sistema *web* intitulado “POSTAGENS” desenvolvido na plataforma acima especificada.

1.4 ESTRUTURA DO TRABALHO

Esta Pesquisa foi organizada da seguinte maneira:

O capitulo 2 apresenta uma revisão sobre computação em nuvem pontuando características e particularidades.

O capitulo 3 trata de forma breve algumas das principais plataformas de desenvolvimento na nuvem disponíveis no mercado atualmente (Amazon *Web Services, MICROSOFT Azure e Google App Engine*).

A plataforma *Google App Engine* é abordado no capitulo 4 onde são abordados o ambiente de execução e os recursos oferecidos pela plataforma.

Os capítulos 5 e 6 apresentam respectivamente a linguagem *Python* e o framework de desenvolvimento web *Django* e sua funcionalidade de modelagem de dados e *Templates*.

No capitulo 7 é mostrado o Desenvolvimento do sistema “POSTAGENS” e no capitulo 8 é feita a tradução do sistema “POSTAGENS” para uma aplicação *Django* apresentando os métodos de integração parcial e depois usando o fork *Django*-nonrel.

2. COMPUTAÇÃO EM NUVEM

De acordo Vecchiola (2009) com o avanço da sociedade moderna, serviços básicos e essenciais são quase todos entregues de uma forma completamente transparente. Serviços de utilidade pública como água, eletricidade, telefone e gás tornaram-se fundamentais para nossa vida diária e são explorados por meio do modelo de pagamento baseado no uso.

Atualmente a mesma idéia tem sido aplicada no contexto da informática, trazendo grandes mudanças na organização da infraestrutura computacional.

Computação em Nuvem tornou-se tendência tecnológica, ao proporcionar plataformas, infraestrutura e serviços sob demanda através do pagamento baseado na utilização.

Esta tecnologia provem serviços com abrangência global, desde o usuário final que envia seu *e-mail* através de um *webmail (SaaS – Software as a Service*), do desenvolvedor que aluga uma plataforma de Desenvolvimento para hospedar sua aplicação (PaaS – Plataform as a Service) até as empresas que terceirizam partes de sua infraestrutura de TI (*IaaS – Infrastructure as a Service*).

Não apenas recursos de computação e armazenamento são entregues sob demanda, mas a maioria dos serviços providos localmente podem ser aproveitada na nuvem.

A computação em nuvem apresenta uma série de vantagens e de estimulantes desafios. Simples tarefas como obtenção, compartilhamento, manipulação e busca de dados exigem um grande volume de recursos para que possam ser executadas.

A Nuvem pode contribuir na resolução dessa problemática à medida que permite disponibilizar recursos de processamento, memória e armazenamento para a utilização imediata, fato que agrega grandes vantagens para organizações e usuários, já que se terceirizam todas as preocupações com

planejamento e manutenção de grandes estruturas de TI, e possibilita que os usuários se concentrem exclusivamente nas regras de negócios.

> A proposta básica da computação em nuvem é que a provisão de recursos computacionais seja de responsabilidade de empresas especializadas ou que seja abstraído o fornecimento dos mesmos em níveis que apenas especialistas venham se preocupar em gerenciá-los e mantê-los, e ainda os mesmo sejam disponibilizados como serviços [Carr, 2008 p.32].

Os recursos precisam ser providos em várias camadas, onde cada camada representa um gênero especifico de recursos que são providos de diferentes formas, o que é de certa forma desafiadora, já que representa uma quebra de paradigma, já que pouco tempo atrás, pessoas e empresas utilizavam exclusivamente os recursos computacionais de forma proprietária, eram responsáveis pela gestão, manutenção e atualização de seus recursos computacionais.

> A nuvem pode ser considerada uma metáfora para a *internet*, sendo baseada em abstrações que ocultam a complexidade de infra-estruturas, onde cada parte é disponibilizada como serviço e hospedada em centros de dados que utilizam *hardware* compartilhado para computação e armazenamento [Buyya, 2009 p.137].

A computação em nuvem possui características que o distingue dos outros paradigmas, por exemplo, a serviço sob demanda, alta escalabilidade, amplitude do acesso e medição do nível do serviço. O NIST *(National Institute of Standards and Technology*) definiu algumas características que descreve o modelo de computação em nuvem, dentre elas estão:

- **Virtualização de recursos:** Muitas tecnologias já estabelecidas possibilitam a virtualização de recursos computacionais, das quais podemos citar, máquinas virtuais, virtualização de redes, de armazenamento e de memória. Com isso se tornou possível a separação dos serviços de infraestrutura dos recursos físicos (*hardware*, redes, periféricos entre outros), o que facilita o tratamento de forma transparente do contexto para as demais camadas, do

tratamento relativo à localização de recursos em uma camada inferior Com isso, torna-se possível o oferecimento de recursos com serviços utilitários, sem que seja necessária a configuração e manipulação direta de *hardware*.

- **Serviços sob demanda:** Conforme necessidade o cliente, pode-se solicitar o aumento ou a diminuição de um ou vários recursos computacionais, tal como tempo de processamento, armazenamento e largura de banda, que são automaticamente disponibilizados, sem a necessidade de interação humana com o provedor de cada serviço. Cada provedor de recursos atende a vários consumidores através de um modelo multi-clientes, alocando dinamicamente recursos físicos e virtuais de acordo com a demanda de cada um.

- **Independência de localização:** Todos os recursos da nuvem devem estar disponíveis através da rede, acessíveis por meio de diversos dispositivos computacionais como telefones celulares, laptops, PDAs. De forma que, a nuvem se torne um ponto de acesso que centraliza as necessidades computacionais dos usuários, ficando disponível em qualquer lugar em todo o tempo. O usuário não conhece a localização física dos recursos computacionais, já que a nuvem pode a seu critério direcionar as solicitações para um *datacenter* com uma carga menor de solicitações ou que tenha uma menor latência em relação a quem solicita.

- **Elasticidade e Escalabilidade:** É a principal característica da computação em nuvem e também a mais inovadora. Elasticidade é a capacidade de disponibilizar e remover recursos computacionais em tempo de execução, independentemente da quantidade solicitada. Esse recurso permite que o prestador de serviços projete seu sistema sem se preocupar com o dimensionamento da infraestrutura, e também evita desperdício de recursos quando se trata de um sistema de uso sazonal, já que podem ser solicitados assim que

necessário em tempo real em períodos de pico, assim como podem ser removidos quando houver pouco uso.

Para os usuários, os recursos são virtualmente ilimitados e podem ser adquiridos em qualquer quantidade, a depender da demanda determina-se a liberação ou aumento dos recursos, de forma rápida transparente e sem intervenção humana.

- **Medição dos Serviços:** Analogamente a serviços de utilidade pública, que devem estar disponíveis a qualquer momento, e que se paga aos provedores em razão da quantidade consumida; a nuvem controla e melhora o uso dos recursos por meio de medições dos serviços providos.

> A monitoração é feita de forma transparente tanto para o servidor quanto para o cliente, onde normalmente são usados contratos referentes aos serviços (SLA – Service Level Agreement) para especificar as características dos serviços, parâmetros de qualidade (QOS – Quality of services) e para determinar os valores que serão cobrados. [Pervez, 2010 p.97]

Os níveis de funcionalidade, disponibilidade, desempenho e outros atributos como penalidades em caso de violação são definidos no SLA.

- **Repositórios de Recursos:** A nuvem é projetada e organizada para atender a múltiplos usuários ao mesmo tempo. Para isso diferentes recursos físicos e virtuais podem ser atribuídos e conFigurados dinamicamente de acordo com a demanda de cada cliente.

2.1 MODELO DE SERVIÇOS

O modelo conceitual de três camadas é o mais citado na literatura, e define um padrão arquitetural para soluções de computação em nuvem. A Figura 2.1 representa o modelo de serviços da computação em nuvem.

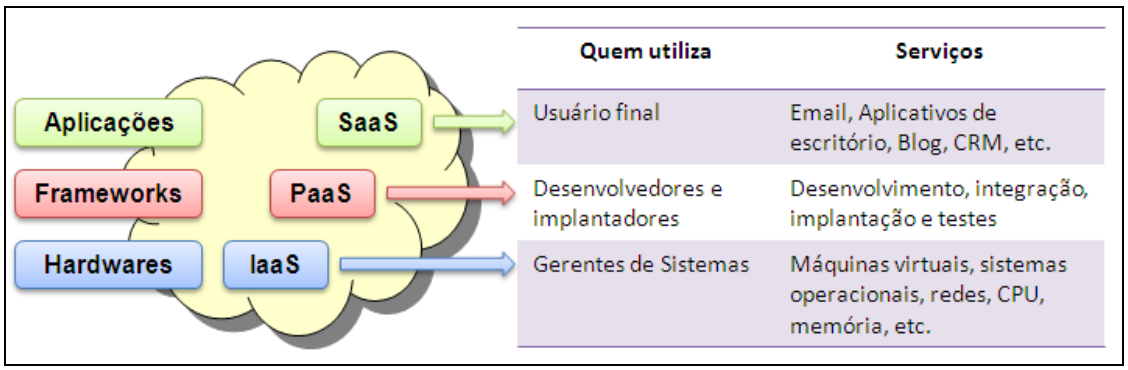


Figura 2.1 Modelo de Serviços da Computação em Nuvem Fonte: (BORGES, 2011)

Na Figura 2.2, observa-se exemplos de aplicações destacadas por tipo de camada.

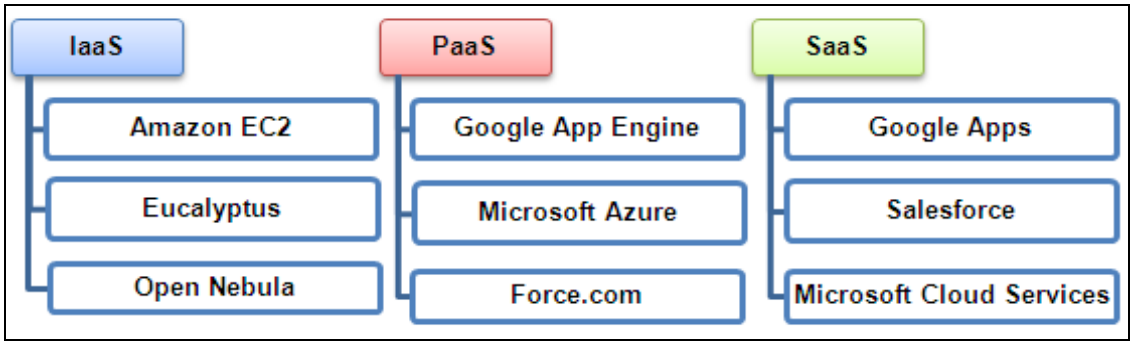


Figura 2.2 Exemplos de Serviço Fonte: (BORGES, 2011)

2.1.1 IAAS INFRAESTRUTURA COM SERVIÇO

É a camada mais inferior do modelo conceitual, composta por plataforma para desenvolvimento, teste, implantação e execução de aplicações, relacionam-se com a capacidade que um provedor tem de oferecer uma infraestrutura de processamento e armazenamento de forma transparente.

Segundo Sousa (2009) seu principal objetivo é tornar mais fácil e acessível o fornecimento de recursos, como servidores, redes, armazenamento e outros que são fundamentais na construção de um ambiente sob demanda podendo ser tanto sistemas operacionais quanto aplicativos.

A camada de infraestrutura se faz possível por meio da virtualização computacional que permite o escalonamento de forma dinâmica dos recursos de acordo com as necessidades das aplicações.

O uso da IaaS reduz investimentos com *hardware*, elimina custos com segurança e manutenção, otimiza o desempenho, libera espaço físico na empresa e traz maior flexibilidade para ampliar e reduzir a capacidade de processamento ou armazenamento de acordo com a sua necessidade.

2.1.2 PAAS – PLATAFORMA COMO SERVIÇO

No modelo conceitual é a camada intermediária, é composta por *hardware* virtual disponibilizado como serviço. Fornecem ambiente de desenvolvimento de *software* com tipos específicos de bancos de dados, serviços de mensagens, serviços de armazenamento de dados, e chamadas ao sistema.

Essa camada facilita a implantação de aplicações sem os custos e complexidades relativos a compra e gerenciamento de *hardware* e *software* que são necessários ao ambiente de desenvolvimento.

Os serviços fornecidos nessa camada são: ferramentas de desenvolvimento de aplicações, testes, implantação, integração de serviços *web*, segurança, integração de banco de dados, persistência, hospedagem. Os quais podem também ser oferecidos como uma solução integrada.

2.1.3 SAAS – SOFTWARE COMO SERVIÇO

É a camada mais externa no modelo conceitual, basicamente composta por aplicações completas ou conjunto de aplicações reguladas por modelos de negócio que permitam customização e que são executados no ambiente da nuvem.

Esses sistemas estão disponíveis através de qualquer navegador *web*, por isso devem estar acessíveis de qualquer lugar a partir dos diversos dispositivos dos usuários.

Tarefas como atualização e adição de novas funcionalidades, podem ser realizadas de forma transparente, sem afetar a usabilidade do sistema

2.1.4 OUTRAS CAMADAS

Alguns conceitos são criados e utilizados para diferenciar um determinado tipo de serviço, como o Banco de Dados como serviço (DaaS - *Database as a Service*) e Testes como serviço (*TaaS- Test as a Service*).

2.2 MODELOS DE IMPLANTAÇÃO

De acordo com as diversas abordagens a respeito da computação em nuvem existem vários modelos de implantação descritos na literatura.

A Figura 2.3 representa o modelo geral de implantação da computação em nuvem.

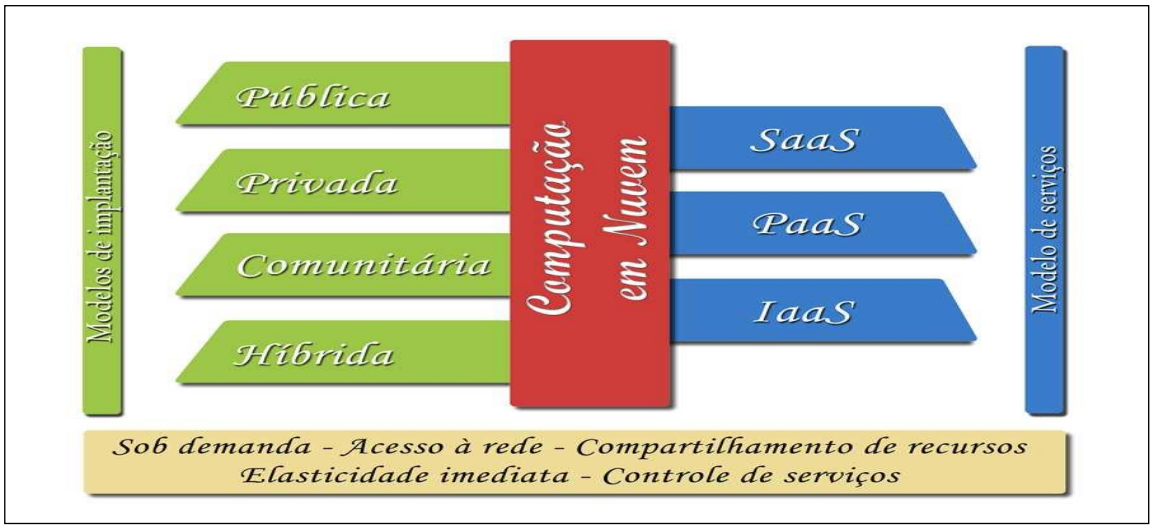


Figura 2.3 Modelo geral de implantação da computação em nuvem Fonte: (BORGES, 2011)

Segundo Rodrigues e Neubauer (2010) no modelo de implantação dependemos das necessidades das aplicações que serão implementadas. A

restrição ou abertura de acesso depende do processo de negócios, do tipo de informação e do nível de visão desejado. Percebemos que certas organizações não desejam que todos os usuários possam acessar e utilizar determinados recursos no seu ambiente de computação em nuvem.

Os modelos de implantação mais descritos são nuvem privada, nuvem pública, Nuvem Comunidade e Nuvem Hibrida.

2.2.1 NUVEM PRIVADA

Modelo no qual a infraestrutura é proprietária, sendo operada por uma única organização. A Figura 2.4 mostra a nuvem privada.

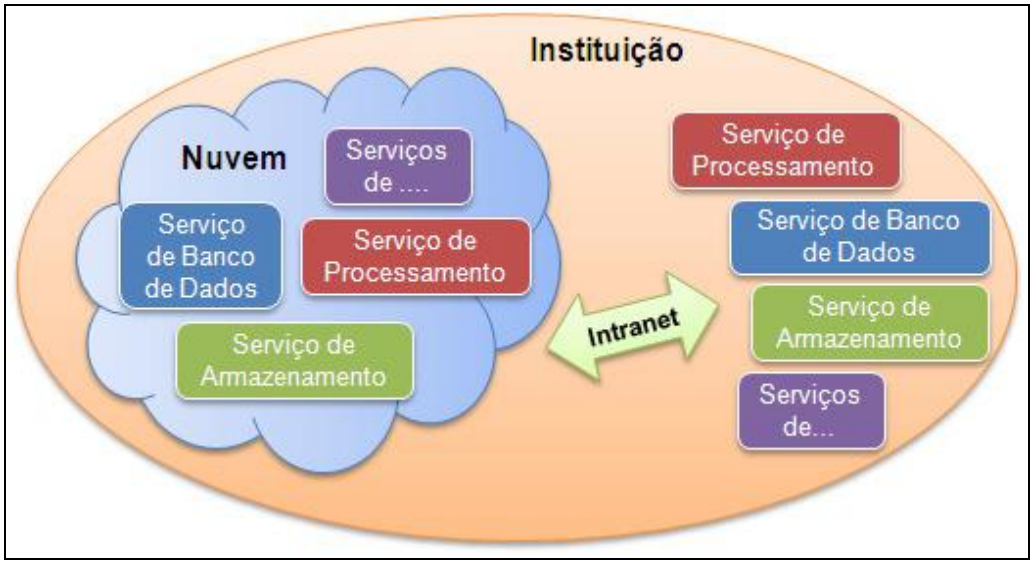


Figura 2.4 Nuvem privada Fonte: (BORGES, 2011)

Segundo Taurion (2009) a característica que diferencia as nuvens privadas é o fato da restrição de acesso, pois a mesma se encontra atrás do *firewall* da empresa, sendo uma forma de aderir à tecnologia, beneficiando-se das suas vantagens, porém mantendo o controle do nível de serviço e aderência às regras de segurança da instituição.

A maior dificuldade para estabelecer uma nuvem privada são os grandes custos de operação, porém, oferecem um controle mais detalhado sobre os vários recursos que constituem a nuvem.

2.2.2 NUVEM PÚBLICA

Nesse modelo a infraestrutura pertence à uma organização que comercializa serviços para o público em geral e que pode ser acessada por qualquer usuário que conheça a localização do serviço.

Nesse tipo de serviço é admitida restrição ao acesso e autenticação. A Figura 2.5 mostra o modelo da nuvem pública.

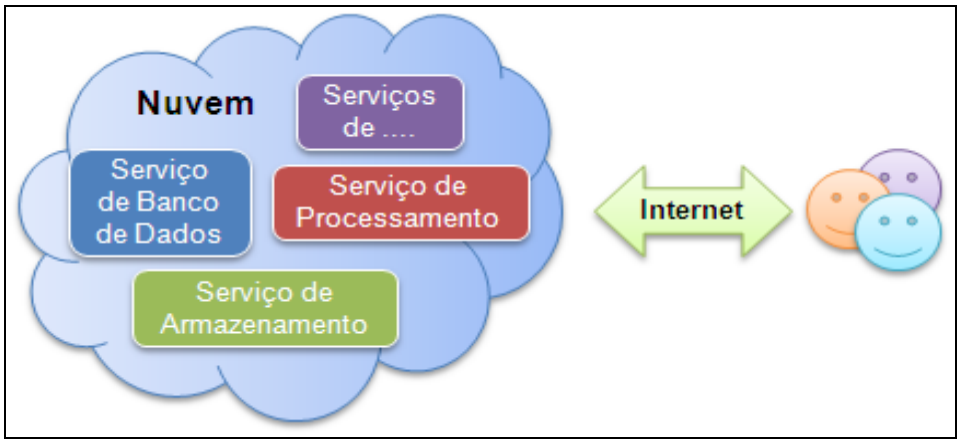


Figura 2.5 Nuvem pública (BORGES, 2011)

Essas nuvens fornecem aos clientes serviços de TI com menor complexidade, onde o provedor assume as responsabilidades de instalação, gerenciamento, disponibilização e manutenção.

De acordo Taurion (2009) esses serviços são oferecidos com conFigurações especificas que abrangem casos mais comuns de uso, logo, não representam a solução mais adequada para processos que exigem maior segurança e restrição.

2.2.3 NUVEM COMUNIDADE.

O modelo nuvem comunidade possui sua infraestrutura compartilhada por diversas organizações com interesses comuns. A Figura 2.6 mostra o modelo nuvem comunidade.

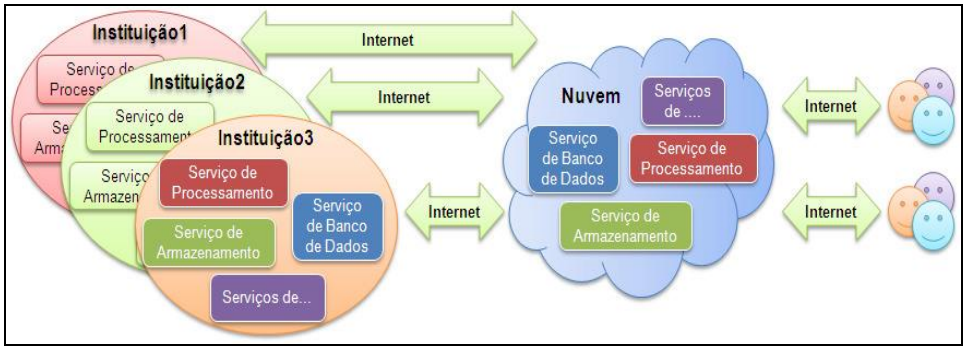
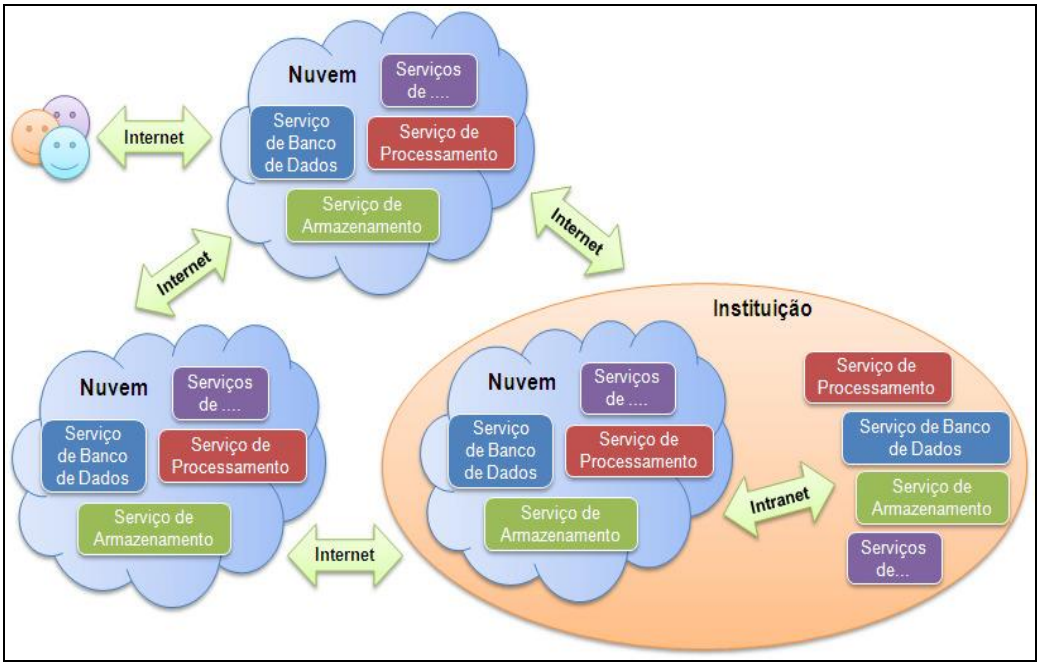


Figura 2.6 Nuvem comunidade (BORGES, 2011)

2.2.4 NUVEM HIBRIDA.

A infraestrutura desta nuvem é composta por pelo menos dois modelos de nuvens, que preservam as características originais de seu modelo, porém são interligadas entre si por meio de tecnologias que permitem a interoperabilidade entre as aplicações. A Figura 2.7 apresenta o modelo nuvem hibrida.



A Figura 2.7 modelo nuvem hibrida Fonte: (BORGES, 2011)

Esse modelo de nuvem pode atender processos críticos e seguros, assim como os secundários para o negócio, como processamento de folha de pagamento de funcionários.

Uma das grandes dificuldades é a de se criar um mecanismo eficaz, capaz de gerenciar e prover a interação entre os diversos tipos de nuvens e disponibilizá-los como se fossem originados de um só fornecedor.

2.3 PAPÉIS NA NUVEM

É muito importante a definição das responsabilidades de cada ator envolvido nas soluções de computação em nuvem. Marinos and Briscoe (2009) classificou os atores de acordo com o papel que desempenham e esta classificação pode ser vista no diagrama abaixo, ela exibe as interações entre os atores e os serviços. A Figura 2.8 retrata o funcionamento do ambiente de nuvem, onde os provedores de serviço gerenciam a infraestrutura, os desenvolvedores criam e disponibilizam os serviços e por fim os consumidores utilizam os serviços através da nuvem.

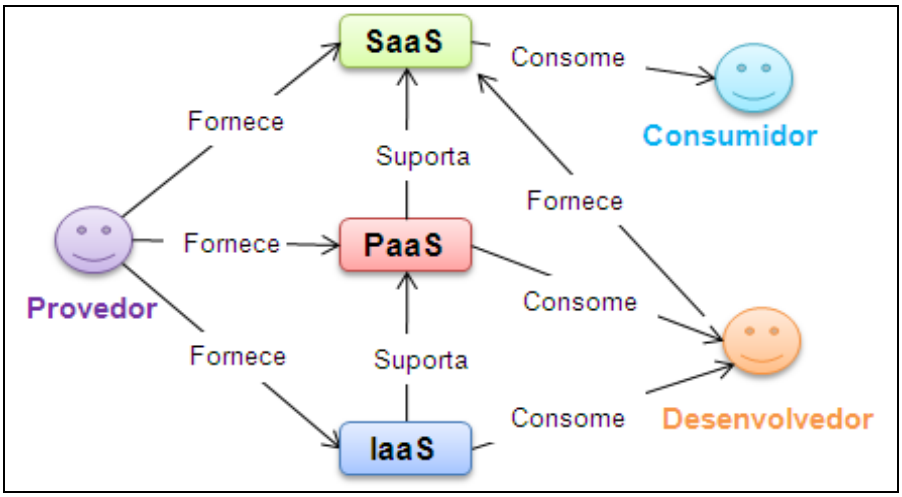


Figura 2.8 Papéis Fonte: (BORGES, 2011)

Ao usuário final não é necessário ter conhecimentos sobre computação em nuvem para utilizar uma aplicação hospedada na nuvem. A Figura 2.9 representa o diagrama de papéis onde os consumidores utilizam os serviços disponíveis através da nuvem.

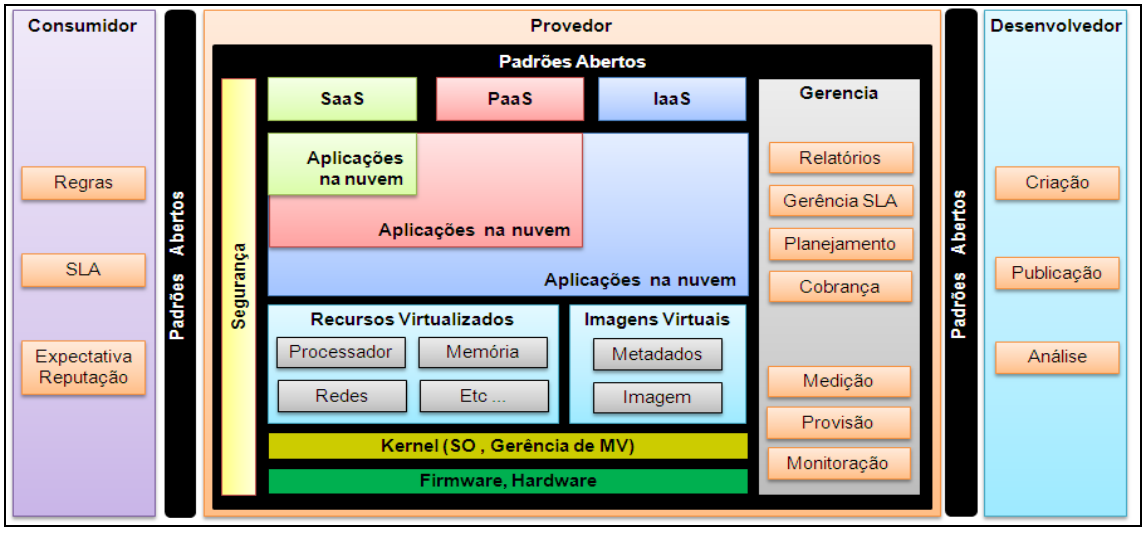


Figura 2.9 Diagrama dos Papéis (BORGES, 2011)

Cabe ao consumidor conhecer a reputação os acordos de nível de serviço e as regras as quais está sujeito.

Ao desenvolvedor é devido o processo de criação publicação de teste de sistemas na nuvem, utilizando as interfaces fornecidas pelos modelos de serviços.

A primeira camada do diagrama do provedor de serviços é o *firmware* e o *hardware* que são base da estrutura da nuvem. Acima disso está o kernel, tanto o sistema operacional quanto o gerenciador da máquina virtual que hospeda a infra-estrutura da nuvem.

Logo acima do Sistema Operacional vemos a camada de virtualização de *hardware*, que suporta os modelos de serviços da computação em nuvem.

O primeiro modelo é a Infraestrutura como serviço que suporta a plataforma e que por sua vez suporta a camada de *software* como serviço.

Trabalhando de forma independente podemos observar o sistema de gerência de nuvem, que é responsável pela geração de medição, a provisão e a monitoração e a provisão de recursos; Além da geração de relatórios gerenciais, o gerenciamento SLA, do planejamento e da cobrança.

A cobrança utiliza os resultados da medição e o planejamento que deve garantir que a demanda seja suprida. A gerência de SLA serve para monitorar se os acordos estão sendo cumpridos e ainda devem ser produzidos relatórios para a análise pelos administradores.

2.4 CENÁRIOS NA NUVEM

Descreveremos a seguir alguns dos possíveis casos de interação no ambiente da nuvem, ilustrando as possibilidades mais comuns.

2.4.1 USUÁRIO FINAL – NUVEM

Neste cenário, o usuário acessa dados ou aplicações na nuvem, como e-mail, redes sociais, armazenamento de arquivos através de uma aplicação sendo executada a partir de um navegador. A Figura 2.10 ilustra este cenário.

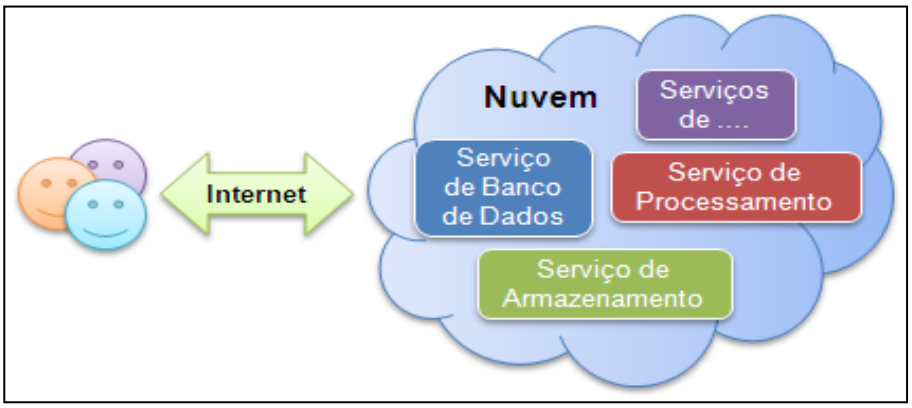


Figura 2.10 Usuário final nuvem Fonte; (BORGES, 2011)

Não existe nenhuma preocupação, sendo que o usuário não tem idéia de como a arquitetura funciona.

2.4.2 ORGANIZAÇÃO – NUVEM USUÁRIO FINAL

Nesse cenário uma empresa utiliza a nuvem para prover dados e serviços para os usuários. Quando o usuário interage com a instituição, a mesma acessa a nuvem para manipular os dados e então retornar os resultados para o usuário requisitante.

Veja a Figura 2.11 para visualizar essa organização

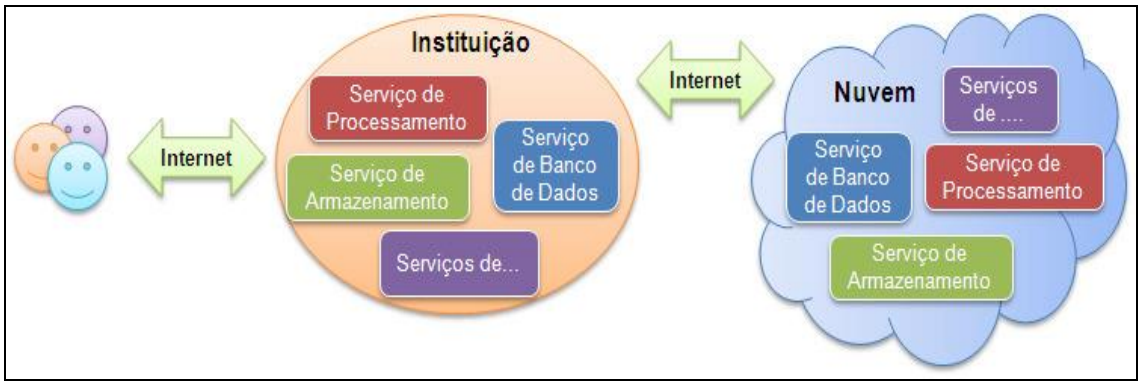


Figura 2.11 Organização nuvem usuário final Fonte: (BORGES, 2011)

2.4.3 ORGANIZAÇÃO – NUVEM

Esta situação representa uma empresa que utiliza os serviços da nuvem para resolver processos internos. Ver Figura 2.12.

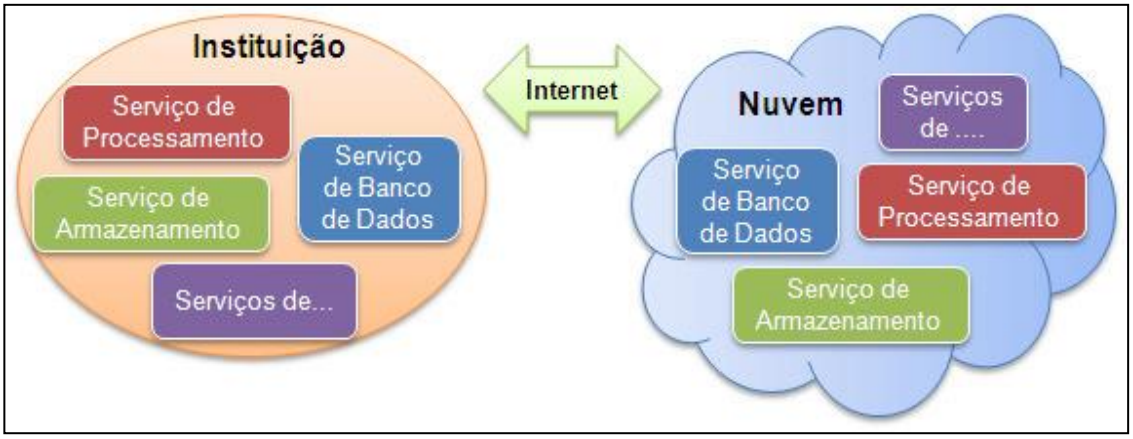


FIGURA 2.12 Organização- Nuvem FONTE (BORGES, 2011)

> Esta provavelmente é a forma de utilização mais comum da computação em nuvem porque proporciona um maior controle pela instituição. Os usos mais comuns envolvem o armazenamento de *backups* na nuvem ou armazenamento de dados raramente utilizados; uso de máquinas virtuais para trazer processadores on-line para momentos de alta demanda de recursos e obviamente o descarte das mesmas quando a demanda diminuir; uso de aplicações SaaS, como e-mail, agenda, etc; e ainda utilização de bases de dados como parte de uma aplicação o que permite um fácil compartilhamento destas bases com parceiros. [Buyya et al. 2009 p 25]

2.4.4 ORGANIZAÇÃO – NUVEM – ORGANIZAÇÃO

Aqui duas organizações utilizam a mesma nuvem, a idéia principal desta proposta é a hospedar recursos na nuvem de modo que as aplicações das empresas possam interoperar. Observe a Figura 2.13.

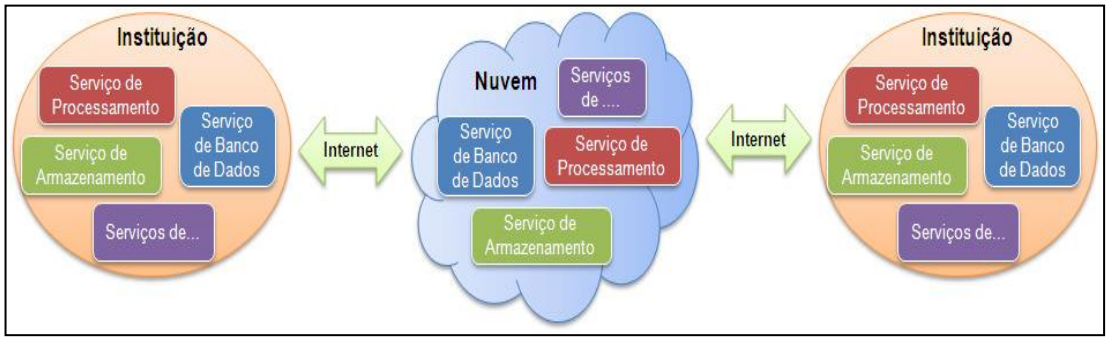


Figura 2.13 Organização nuvem – organização Fonte(BORGES, 2011)

3. PLATAFORMAS DE DESENVOLVIMENTO NA NUVEM

A computação em nuvem fez com que o mercado de Tecnologia da informação (TI) repensasse suas estratégias. Estas estratégias variam de empresa para empresa, mas de maneira geral, as mais tradicionais em TI se movimentam para concentrar seus esforços nos segmentos que conhecem bem, o corporativo, seja de grande porte ou mesmo, em alguns casos, o segmento de pequenas e médias empresas. Por sua vez, as *startups*[4] se concentram nos usuários finais e empresas de pequeno a médio porte.

Neste capitulo vamos detalhar um pouco mais as estratégias e ofertas de alguns dos principais provedores de Computação em Nuvem, que variam de *startups* como 3Tera, Joyent, Nirvanix e Gogrid, a empresas já consolidadas no mercado de TI, como *HP, Dell, Oracle, IBM, SalesForce, Microsoft, Google, Amazon* entre outras.

3.1 AMAZON WEB SERVICES (AWS)

Um exemplo já bem conhecido de computação em nuvem são os serviços da *Amazon*, que criou uma subsidiária chamada *Amazon Web Services* para ofertar serviços de computação em Nuvem

O *Amazon Web Services* (AWS) é um ambiente de computação em nuvem disponível através de serviços *web* e com características de escalabilidade, disponibilidade, elasticidade e desempenho para aplicações executadas nesse ambiente. O AWS é composto por um conjunto de sistemas que fornecem os diferentes modelos de serviço. Basicamente são ofertados quatro serviços. O *EC2*[5], para alugar máquinas virtuais *Linux*, nas quais o usuário pode alugar grandes quantidades de máquinas virtuais Linux, o *S3*

[4] Startups é um modelo de empresa jovem embrionária ou ainda em fase de constituição e implantação e organização de suas operações

[5] EC2 – *Elastic Computing Cloud*

serviço de armazenamento em nuvem, o *SimPleDB* oferta de *Database-as-a-Service*[6] e o SQS (Simple Queue Service) usado para serviços de enfileiramento de mensagens. A idéia é que os usuários possam operar seus negócios sem ter a necessidade de investir em infra-estrutura, como servidores e armazenamento.

3.2 MICROSOFT AZURE

É uma plataforma para a implementação de computação em nuvem que oferece um conjunto especifico de serviços para desenvolvedores. Esta plataforma pode ser usada por aplicações em execução na nuvem ou fora dela. A plataforma *Azure* é formada pelos sistemas operacional *Windows Azure* e um conjunto de outros serviços.

O *Windows Azure* é o sistema operacional para serviços na nuvem que é utilizado para o desenvolvimento, hospedagem e gerenciamento dos serviços dentro do ambiente *Azure.*

3.3 *GOOGLE APP ENGINE*

Segundo BARROS (2009) O *Google App Engine* é uma plataforma para o desenvolvimento de aplicações web escaláveis que são executadas na infraestrutura do *Google*. Ele fornece um conjunto de APIs e um modelo de aplicações que permite aos desenvolvedores utilizarem serviços adicionais fornecidos pelo *Google*, como o *e-mail* e autenticação.

De acordo com o modelo de aplicação previsto, os desenvolvedores podem criar aplicações *Java* e *Python* e utilizar diversos recursos tais como: armazenamento, transações, ajuste e balanceamento de carga automáticos, ambiente de desenvolvimento local e tarefas programadas.

Para Chang (2006) o *Google App Engine* possui um serviço de

[6]Database-as-a-Service, ou banco de dados como serviço, com o uso da estrutura de database distribuída e remota como se fosse local

armazenamento baseado no *BigTable*, um sistema distribuído de armazenamento de dados em larga escala.

As aplicações desenvolvidas para o *App Engine* serão executadas no *Google*, que realiza automaticamente, caso necessário, o dimensionamento. O *Google* oferece um serviço gratuito limitado e a opção de contratar um serviço profissional.

O modelo de Computação em nuvem ganhou espaço na mídia pelos exemplos pioneiros de empresas como o *Google*. Uma nuvem computacional que todos nós usamos. Uma simples consulta no *Google* demanda acesso a milhares de *megabytes* e consome dezenas de bilhões de ciclos de processador. Em seus *data centers* o *Google* mantém uma grande fração da *Internet*, e esta cópia está sendo continuamente atualizada através de *software spiders* que percorrem a rede, link por link, vasculhando o conteúdo das suas bilhões de páginas. Construir uma infraestrutura computacional que permita milhares de *queries* por segundo, e completar o processo em poucos segundos, nos moldes tradicionais, iria requerer inúmeros supercomputadores de centenas de milhares de dólares cada. o *Google* resolveu o problema, a um custo bem menor, agrupando centenas de milhares de servidores *Intel/Linux* em seus *data centers* espalhados pelo mundo.

A aplicação do buscador *Google* é por natureza altamente paralelizável, permitindo plena exploração do conceito de nuvem. Diferentes consultas rodam em diferentes processadores e o índice é particionado de modo que uma única query[7] possa também usar múltiplos processadores em paralelo.

A execução de uma consulta consiste basicamente em duas etapas. Na primeira, os servidores de índice consultam uma lista invertida que mapeia cada palavra da consulta a uma lista de documentos que satisfazem a pesquisa. Adicionalmente é computado um fator de relevância, que vai determinar a ordem de visualização nas páginas de saída exibidas ao usuário.

Os índices são muito grandes. Constituem-se em arquivos de muitos *terabytes*. O índice é dividido em blocos, que podem ser pesquisados pelos

[7] query: consulta realizada em um banco de dados

servidores de índice simultaneamente. O resultado é uma pesquisa rápida e eficiente.

Depois de gerada a lista de documentos que satisfaçam a pesquisa, os documentos em si devem ser acessados. Cada documento deve ser lido e recuperado em seu titulo e resumo, como aparecem nas páginas de saída do navegador. Os documentos são espalhados e os servidores de documentos podem ser acionados simultaneamente, fazendo uma rápida recuperação destes documentos.

O processo de busca do sistema do *Google* possui características que permite explorar a potencialidade do paralelismo: já que os acessos que realizam apenas leitura no banco podem ser efetuados em paralelo, já que cada bloco de índice ou de documento não possui relação com os demais.

A conclusão a que se chega é que o *Google* consegue realizar pesquisas rápidas, com uma infraestrutura em TI baseada em uma grande quantidade de servidores customizados de baixo custo ao invés de usar apenas um supercomputador que oferecesse a mesma capacidade computacional o que acaba diminuindo os custos e aumentado o paralelismo do sistema.

Apesar de num primeiro momento da se a entender que operar milhares de máquinas ao invés de poucas tende a aumentar os custos de gerenciamento, o fato das nuvens executarem poucas aplicações dedicadas e as configurações serem mantidas de forma homogênea, faz com que seja possível manter o gerenciamento.

O *Google* expandiu o uso das suas nuvens para acomodar outros de seus serviços, como *GMAIL, Google Maps, Youtube e o Google App Engine*[8] que é uma plataforma para desenvolvimento de aplicações para a nuvem do Google.

[8]Google App-Engine - http://code.google.com/appengine/

O *Google App Engine* permite ao desenvolvedor de *software* criar sua aplicação sem necessidade de possuir servidores e *software* específicos. Todo o processamento é feito na nuvem e a aplicação é projetada para rodar especificamente na nuvem do *Google*. O ambiente de desenvolvimento consiste em um SDK[9] que contêm um servidor que simula localmente a infraestrutura da nuvem do *Google App Engine*.

[9]SDK - (Software Development Kit)

4. GOOGLE APP ENGINE

Google App Engine (GAE) é um serviço de hospedagem de aplicações web que também pode servir para armazenar conteúdo estático como imagens documentos e páginas *web* estáticas.

> Uma das particularidades do *App Engine* está no fato de que ele foi projetado com foco na escalabilidade, portanto ele é ideal para aplicações que possuem muitos usuários simultâneos. As aplicações que são escritas para executar na infraestrutura do *Google App Engine* são nativamente escalonáveis (SANDERSON, 2010, p. 15).

O *App Engine* aloca mais recursos automaticamente para as aplicações e gerencia o uso destes recursos. A aplicação não precisa saber sobre a quantidade de recursos que esta usando.

Segundo SANDERSON (2009) diferentemente da hospedagem de *sites* tradicional ou dos servidores auto-gerenciados, no GAE só se paga pelos recursos que foram usados. Esses recursos são trafego de rede mensal, tempo de CPU, armazenamento mensal.

O *Google App Engine* facilita a criação de aplicativos que podem ser executados de forma confiável mesmo sob uma carga pesada e com grandes quantidades de dados. O *Google App Engine* inclui os seguintes recursos:

- Serviço de *web* dinâmico, com suporte completo a tecnologias de web comuns;
- Armazenamento persistente com consultas, classificação e transações;
- Ajuste e balanceamento de carga automáticos;
- APIs para autenticação de usuários e envio de *e-mail* usando contas do Google;
- Um ambiente de desenvolvimento local com todos os recursos, simulando o *Google App Engine* em seu computador;
- Tarefas programadas para disparar eventos em horários específicos e em intervalos regulares;

O aplicativo pode ser executado em um dos dois ambientes de

execução: *Java* e *Python*. Cada ambiente oferece os protocolos padrões e tecnologias comuns para o desenvolvimento de aplicativos da web.

O *Google App Engine* pode ser dividido em três partes: O ambiente de execução, o sistema de armazenamento e as funcionalidades adicionais.

4.1 O AMBIENTE DE EXECUÇÃO

As aplicações que rodam no *Google App Engine* tem a finalidade básica de responder à requisições *web*. Uma requisição *web* começa quando um cliente contata a aplicação com uma requisição *HTTP*. Ao receber uma requisição ele identifica a aplicação pelo seu domínio e a encaminha.

> Os aplicativos são executados em um ambiente seguro chamado Sandbox, que fornece acesso limitado ao sistema operacional. Essas limitações permitem que o *Google App Engine* distribua solicitações de web do aplicativo entre diversos servidores, iniciando e interrompendo os servidores para atender às demandas de tráfego. (SANDERSON,2009,p. 2)

O *Sandbox* isola o aplicativo em seu próprio ambiente seguro e confiável, independentemente de *hardware*, sistema operacional e localização física do servidor da web.

As principais limitações são:

- Um aplicativo pode acessar outros computadores na internet somente por meio dos serviços de obtenção de *URL* e de *e-mail* fornecidos. Outros computadores podem se conectar ao aplicativo somente fazendo solicitações *HTTP* (ou HTTPS) nas portas padrão.

- Um aplicativo não pode gravar no sistema de arquivos. Um aplicativo pode ler arquivos, mas somente os enviados junto com o código do aplicativo. O aplicativo deve usar o armazenamento de dados, o cache de memória ou outros serviços do *Google App Engine* para todos os dados que persistirem entre as solicitações.

- O código do aplicativo é executado somente em resposta a uma solicitação da *web* ou um trabalho do *cron*, e deve retornar dados de resposta em 30 segundos, independentemente do caso. Um manipulador de solicitação não pode gerar um subprocesso nem executar o código após o envio da resposta.

4.2 O ARMAZENAMENTO DE DADOS

O *Google App Engine* fornece o serviço de armazenamento de dados distribuído que contém um mecanismo de consultas e transações. Assim como o servidor da web distribuído cresce proporcionalmente ao tráfego, o armazenamento de dados distribuído cresce à medida que os dados aumentam.

> O armazenamento de dados do *Google App Engine* não é um banco de dados relacional tradicional. Os objetos de dados, ou entidades, têm um tipo e um conjunto de propriedades. As consultas podem recuperar entidades de um tipo determinado, filtradas e classificadas segundo os valores das propriedades. Os valores das propriedades podem ser de qualquer um dos tipos de valor de propriedade suportados. (SANDERSON, 2009, p. 110)

As entidades do armazenamento de dados não possuem esquema. A estrutura das entidades de dados é fornecida e aplicada pelo código do seu aplicativo. O aplicativo também pode acessar o armazenamento de dados diretamente para aplicar as mudanças necessárias.

O armazenamento de dados é altamente consistente e usa o controle de concorrência otimista. Uma atualização de entidade ocorre em uma transação com um número fixo de tentativas, caso outros processos estejam tentando atualizar a mesma entidade simultaneamente. O aplicativo pode executar diversas operações de armazenamento de dados em uma única transação, sendo que todas terão sucesso ou falharão, assegurando assim a integridade dos seus dados.

O armazenamento de dados implementa transações por toda a sua rede distribuída usando "grupos de entidades". Uma transação manipula entidades

dentro de um único grupo. As entidades do mesmo grupo são armazenadas juntas, para uma execução de transações eficiente. O aplicativo pode atribuir entidades aos grupos quando elas forem criadas.

4.3 O AMBIENTE DE EXECUÇÃO EM *PYTHON*

Com o ambiente de execução em *Python* do *Google App Engine*, é possível implementar os aplicativo usando a linguagem de programação *Python* e executá-lo em um interpretador otimizado de *Python*. O *Google App Engine* inclui APIs avançadas e ferramentas para desenvolvimento de aplicativos da web em *Python*, incluindo uma API de modelagem de dados avançada, uma estrutura para aplicativos da web de fácil uso e ferramentas para gerenciar e acessar os dados do seu aplicativo. Pode se também aproveitar uma grande variedade de bibliotecas e estruturas maduras para o desenvolvimento de aplicativos da web em *Python*, como o *Django* e outros *frameworks*.

> O ambiente Python inclui a biblioteca Python padrão. Nem todos os recursos da biblioteca podem ser executados no ambiente do Sandbox. Por exemplo, uma chamada para um método que tenta abrir um soquete ou gravar em um arquivo emitirá uma exceção. Diversos módulos da biblioteca padrão cujos recursos principais não são suportados pelo ambiente de execução foram desativados, e se os códigos que importam esses recursos forem utilizados, será emitido um erro.(SANDERSON,2009, p.23)

Uma outro limitação encontrada no *Sandbox* é que o código dos aplicativos para o ambiente *Python* deve ser criado exclusivamente em *Python.* As extensões criadas na linguagem C não são suportadas.

O ambiente *Python* fornece APIs abrangentes de *Python* para o armazenamento de dados, contas do *Google*, serviços de obtenção de URL e e-mail. O *Google App Engine* fornece uma estrutura *Python* simples de aplicativo da *web* denominada *webapp*, que facilita a criação de aplicativos *web.*

Pode-se fazer *upload* de outras bibliotecas de terceiros com seu aplicativo, desde que elas sejam implementadas em *Python* puro e não exijam

nenhum módulo de biblioteca padrão não suportado.

4.4 SERVIÇOS ADICIONAIS

O *Google App Engine* fornece diversos serviços que permitem executar operações comuns ao gerenciar os aplicativos.

4.4.1 AUTENTICAÇÃO NAS CONTAS DO GOOGLE

O *Google App Engine* suporta a integração de um aplicativo com as Contas do *Google* para autenticar um usuário. Os aplicativos podem permitir que um usuário faça *login* com uma conta do *Google* e acesse o endereço de e-mail e o nome de exibição associados à conta. O uso de contas do Google permite que o usuário comece a usar os aplicativos mais rapidamente, pois não será mais necessário criar uma nova conta. Isso economiza o esforço de implementar um sistema de contas de usuário somente para o seu aplicativo.

A API de usuários também pode informar ao aplicativo se o usuário atual é um administrador registrado do aplicativo. Isso facilita a implementação de áreas do seu site restritas a administradores.

4.4.2 OBTENÇÃO DE URLS

Os aplicativos podem acessar recursos da internet, como serviços da web ou outros dados, usando o serviço de obtenção de URL[10] do *Google App Engine*. O serviço de obtenção de URL recupera recursos da web usando a mesma infraestrutura de alta velocidade do *Google* que recupera páginas web de muitos outros produtos do *Google.*

[10] **URL:** *Uniform Resource Locator*), em português *Localizador-Padrão de Recursos*, é o endereço de um recurso disponível em uma rede

4.4.3 MENSAGENS

Os aplicativos podem enviar mensagens de *e-mail* usando o serviço de mensagens do *Google App Engine.* O serviço de mensagens usa a infraestrutura do *Google* para enviar mensagens de *e-mail*.

4.4.4 CACHE DE MEMÓRIA

O serviço de cache de memória fornece ao seu aplicativo um cache de memória essencial de alto desempenho, que pode ser acessado por diversas instâncias do seu aplicativo. O cache de memória é útil para dados que não precisam dos recursos de persistência e transações do armazenamento de dados, como dados temporários ou copiados do armazenamento de dados para o cache para um acesso de alta velocidade.

4.4.5 - MANIPULAÇÃO DE IMAGENS

O serviço de imagens permite que seu aplicativo manipule imagens. Com esta API, você pode redimensionar, cortar, girar e inverter imagens nos formatos *JPEG* e *PNG*.

4.4.6 TAREFAS PROGRAMADAS

O *App Engine Cron Service* permite programar tarefas para serem executadas em intervalos regulares.

4.4.7 SUPORTE AO DESENVOLVEDOR

Os SDKs (kits de desenvolvimento de software) do *Google App Engine* para *Java* e *Python* incluem um aplicativo do servidor da web que emula todos os serviços do *Google App Engine* localmente. Cada SDK inclui todas as APIs e bibliotecas disponíveis no *Google App Engine*. O servidor da *web* também

simula o ambiente seguro do *Sandbox*, incluindo verificações de tentativas de acesso não permitido aos recursos do sistema no ambiente de execução do *Google App Engine*.

Cada SDK também inclui uma ferramenta para enviar o seu aplicativo para o *App Engine*. Depois de criar o código e os arquivos estáticos e de configuração dos aplicativo, ao executar a ferramenta para enviar os dados. Ela solicita o endereço de e-mail e a senha de sua conta do *Google*.

Ao criar uma nova versão principal de um aplicativo já sendo executado no *Google App Engine*, pode-se enviá-lo como uma nova versão. A versão antiga continuará servindo os usuários até que o desenvolvedor mude para a nova versão. Pode-se testar a nova versão no *Google App Engine* enquanto a versão antiga ainda está sendo executada.

O SDK para *Python* é implementado em *Python* puro e executado em qualquer plataforma com *Python* 2.5, incluindo Windows, *Mac OS X e Linux*.

O Console de administração é a interface baseada na web para gerenciar os aplicativos sendo executados no *Google App Engine*. Ele deve ser usado para criar novos aplicativos, configurar nomes de domínio, alterar a versão ativa do aplicativo, examinar os *logs* de acessos e de erros e navegar no armazenamento de dados de um aplicativo.

4.4.8 COTAS

Pode-se registrar até 10 aplicativos por conta do desenvolvedor. Cada aplicativo recebe recursos dentro dos limites.

> A criação de um aplicativo do *Google App Engine* não é apenas fácil, é gratuita também. Você pode criar uma conta e publicar um aplicativo que as pessoas usarão imediatamente sem nenhum custo ou compromisso. Um aplicativo de uma conta gratuita pode usar até 500 MB de armazenamento e até 5 milhões de visualizações de página por mês. Quando estiver pronto para mais, você pode ativar o faturamento, definir um orçamento máximo diário e alocar o seu orçamento para cada recurso de acordo com as suas necessidades. (Sanderson, 2009, p. 113)

Alguns recursos impõem limites não relacionados a cotas para proteger a estabilidade do sistema. Por exemplo, quando um aplicativo é chamado para servir uma solicitação da web, ele deve emitir uma resposta dentro de 30

segundos. Se o aplicativo demorar muito, o processo será encerrado e o servidor retornará um código de erro ao usuário. O tempo de espera de solicitação é dinâmico e pode ser reduzido para poupar os recursos caso um manipulador de solicitação chegue ao tempo limite com muita freqüência.

Outro exemplo de limite de serviço é o número de resultados retornados por uma consulta. Uma consulta pode retornar no máximo 1.000 resultados. As consultas que retornariam mais resultados retornam somente o número máximo permitido. Nesse caso, uma solicitação executando essa consulta provavelmente não retornará uma solicitação antes do tempo limite, mas o limite está conFigurado para poupar recursos do armazenamento de dados.

4.5 SOLICITAÇÃO CGI

Quando o *Google App Engine* recebe uma solicitação da web para seu aplicativo, ele chama o *script*[11] manipulador correspondente ao URL, conforme descrito no arquivo de configuração app.yaml do aplicativo. O *Google App Engine* utiliza o padrão CGI para comunicar os dados da solicitação ao manipulador e receber a resposta.

O *Google App Engine* utiliza diversos servidores da web para executar seu aplicativo e ajusta automaticamente o número de servidores sendo usados para manipular as solicitações de forma confiável. Uma determinada solicitação pode ser direcionada a qualquer servidor e este pode não ser o mesmo que manipulou uma solicitação anterior do mesmo usuário.

O servidor determina qual *script* de manipulador em *Python* deve ser executado ao comparar o URL da solicitação aos padrões de URL no arquivo de configuração do aplicativo. Em seguida, ele executa o manipulador em um ambiente CGI preenchido com os dados da solicitação. Conforme descrito no padrão CGI, o servidor coloca os dados da solicitação em variáveis de ambiente e no fluxo de entrada padrão. O *script* executa as ações apropriadas à solicitação e, em seguida, prepara uma resposta e a coloca no fluxo de saída padrão.

[11] script: sequência de comandos

> A maioria dos aplicativos utiliza uma biblioteca para analisar as solicitações CGI e retornar respostas CGI, como o módulo CGI da biblioteca Python padrão ou uma estrutura da web que conheça o protocolo CGI (como o webapp). Você pode consultar a documentação de CGI para obter mais detalhes sobre as variáveis de ambiente e o formato dos dados do fluxo de entrada. (LUTZ, 2009, p.15)

4.6 RESPOSTAS

O *Google App Engine* coleta todos os dados gravados pelo *script* do manipulador de solicitação no fluxo de saída padrão e aguarda a saída do *script*. Quando o *script* é encerrado, todos os dados de saída são enviados ao usuário.

O *Google App Engine* não suporta o envio de dados ao navegador do usuário antes de sair do manipulador. Alguns servidores da web usam essa técnica para "transmitir" dados para o navegador do usuário por um período de tempo em resposta a uma única solicitação. O *Google App Engine* não oferece suporte a essa técnica de transmissão.

> Se o cliente envia cabeçalhos *HTTP* com a solicitação, indicando que o cliente pode aceitar conteúdo compactado (através de gzip), o *Google App Engine* compacta os dados de resposta automaticamente e anexa os cabeçalhos de resposta apropriados. Os cabeçalhos de solicitação `Accept-Encoding` e `User-Agent` são usados para determinar se o cliente pode receber respostas compactadas de forma confiável. Os clientes personalizados podem forçar a compactação do conteúdo, especificando os cabeçalhos `Accept-Encoding` e `User-Agent` com valor "*gzip*". (Sanderson, 2010, p. 60)

4.7 SANDBOX

Para permitir que o *Google App Engine* distribua solicitações para aplicativos em diversos servidores da web e para impedir a interferência de um aplicativo em outro, o aplicativo executa em um ambiente *Sandbox* restrito. Neste ambiente, o aplicativo pode executar código, armazenar e consultar dados no armazenamento de dados do *Google App Engine*, utilizar o e-mail do

Google App Engine, obter URL e serviços de usuários e examinar a solicitação de web do usuário e preparar a resposta.

Um aplicativo do *Google App Engine* não pode:

- Gravar no sistema de arquivos. Os aplicativos devem utilizar o armazenamento de dados do *Google App Engine* para armazenar dados persistentes. É permitida a leitura do sistema de arquivos e todos os arquivos que foram enviados com o aplicativo estão disponíveis.
- Abrir um soquete ou acessar diretamente outro host. Um aplicativo pode utilizar o serviço de obtenção de URL do *Google App Engine* para fazer solicitações HTTP[12] e HTTPS para outros hosts nas portas 80 e 443, respectivamente.
- Gerar um subprocesso ou linha. Uma solicitação da web a um aplicativo deve ser manipulada em um único processo, dentro de alguns segundos. Os processos que demoram muito tempo para responder são encerrados, para evitar sobrecarregar o servidor da web.
- Fazer outros tipos de chamadas do sistema.

[12] HTTP: HyperText Transfer Protocol é um protocolo de aplicação responsável pelo tratamento de pedidos e respostas entre cliente e servidor na World Wide Web.

5. LINGUAGEM PYTHON

Python é uma linguagem de programação de alto nível, interpretada, interativa, orientada à objetos e de tipagem dinâmica e forte. Atualmente possui um modelo de desenvolvimento comunitário e aberto gerenciado pela *Python Software Foundation*.

Apesar de várias partes da linguagem possuírem padrões e especificações formais, a linguagem como um todo não é formalmente especificada. O padrão de implementação é o CPython, o que significa que foi desenvolvida a partir da linguagem C, mas existe tambem as variantes *Jpython* e *IronPython* que são reimplementações do *Python* produzidas em Java .e .Net respectivamente.

A linguagem foi projetada com a filosofia de enfatizar a importância do esforço do programador sobre o esforço computacional. Prioriza a legibilidade de código sobre a velocidade. Combina uma sintaxe concisa e clara com os recursos poderosos de sua biblioteca padrão e por módulos e *frameworks* desenvolvidos por terceiros.

> *Python* é uma linguagem de programação de propósito geral que permite que você trabalhe mais rapidamente e integre seus sistemas de forma mais eficaz. Garantindo assim ganhos imediatos de produtividade e menores custos de manutenção".
> *Lutz & Ascher* (2007 p 13)

5.1 EXEMPLOS DA LINGUAGEM *PYTHON*

A Figura 5.1 e 5.2 possuem exemplos de código que demonstram a simplicidade da linguagem *Python*. Primeiro com o algoritmo Fibonacci usando iteração de listas, e logo depois com o algoritmo do calculo de fatorial de um número.

```
a = 1
a = 1
FIB = input('Informe o numero: ')
for i in range(FIB):
        a,b = b,a+b
print "Fibonacci de ", FIB,"é: ",b
```

Figura 5.1 Fibonacci

```
numero = int(raw_input('Digite um numero: '))
resultado = 1
for x in range(1,numero+1):
resultado = x * resultado
print "Fatorial de ",numero,"é: ", resultado
```

Figura 5.2 Fatorial

O *Google App Engine* executa o código do seu aplicativo *Python* usando um interpretador de *Python* pré-carregado em um ambiente seguro do "*Sandbox".* O aplicativo recebe as solicitações da web, realiza o trabalho e envia as respostas ao interagir com esse ambiente.

Todo código feito para o ambiente de execução deve ser *Python* puro, sem incluir qualquer extensão C ou outro código que necessite de compilação.

O ambiente inclui a biblioteca *Python* padrão. Alguns módulos foram desativados, pois suas funções essenciais não são suportadas pelo *Google App Engine*, como redes ou gravação no sistema de arquivos. Além disso, o módulo **`os`** está disponível, porém com os recursos não suportados desativados. Uma tentativa de importar um módulo ou utilizar um recurso não suportado emitirá uma exceção.

6. FRAMEWORK DJANGO

A cada dia que passa, a funcionalidade e os recursos dos sistemas produzidos evoluem, ao mesmo tempo em que requerido uma maior agilidade no processo de desenvolvimento dos mesmos. Desenvolver software em um curto período de tempo sempre foi um requisito desejável para programadores e empresas.

Vários requisitos, padrões e técnicas são criados diariamente, com a finalidade de aumentar a produção e melhorar o processo de construção de sistemas. Como resultado, foram criados diversas ferramentas e frameworks, vários processos e rotinas repetitivas foram automatizados por eles e o tempo gasto no processo de implementação de fato diminuiu.

Em geral, um framework, assim como web frameworks, proporcionam uma infraestrutura de programação para aplicações, de forma que o projetista possa se concentrar na escrita de código eficiente e de fácil manutenção. Mas ainda há diversos pontos a melhorar, ou pelo menos para facilitar o processo de produção de código de forma automatizada, como por exemplo geração automática de código nativo de um framework a partir do esquema de um diagrama de classes. Os frameworks utilizados atualmente, seguem o padrão MVC, que é um padrão de arquitetura de aplicações que visa separar a lógica da aplicação (Model), da interface do usuário (View) e do fluxo da aplicação (Controller). O uso deste tipo de framework permite que a mesma lógica de negócios possa ser acessada e visualizada por várias interfaces. Neste trabalho adotamos o framework *Django*, que utiliza o padrão MVC e também sua arquitetura de desenvolvimento estimula o desenvolvedor a aproveitar ao máximo o código feito evitando assim repetição.

Django[13] é um framework de desenvolvimento rápido para a web, escrito em *Python*, que utiliza o padrão MVC (model-view-controller). Foi criado originalmente como um sistema para gerenciar um site jornalístico na cidade de Lawrence, no Kansas. Tornou-se um projeto de código aberto e foi publicado

[13]http://www.djangoproject.com/

sob a licença BSD[14] em 2005. O nome *Django* foi inspirado no músico de jazz *Django* Reinhardt.

> Segundo (Holavaty 2009 p.49), Django estimula o desenvolvimento rápido e limpo que utiliza o padrão MVC[15].Ele utiliza o princípio DRY (Don't Repeat Yourself), onde faz com que o desenvolvedor aproveite ao máximo o código já feito, evitando a repetição.

Neste capitulo serão abordadas as principais características do *framework* como *Controllers, Templates e Models além* do fluxo de execução de uma requisição no *Django*.

6.1 PRINCIPAIS CARACTERÍSTICAS

Cada aplicação que você escreve no *Django* consiste de um pacote *Python*, que seguirá uma certa convenção. O *Django* vem com um utilitário que gera automaticamente a estrutura básica de diretório de uma aplicação, então você pode se concentrar apenas em escrever código em vez de ficar criando diretórios.

Através do ORM[16] do *Django*, a modelagem de dados é definida através de classes *Python* nomeadas *Models*. Criando uma camada de abstração a mais para o programador. Independente do SGBD[17] escolhido será possível manipular os dados, sem que seja necessária a criação de *scripts* de código SQL para cada um deles.

O uso de *Models* ao invés de SQL permite ao *Django* a geração automática de uma interface de administração completa para o sistema. O que aumenta a rapidez no desenvolvimento dos sistemas web.

> O Django, como arcabouço para o desenvolvimento de aplicações *web*, mostra-se vantajoso em relação ao desacoplamento de aplicações do projeto, à pequena quantidade de código que será escrita, ao rápido desenvolvimento de aplicações, à filosofia DRY ("*Don't Repeat Yourself*"), que por utilizar uma linguagem totalmente

[14]http://www.opensource.org/licenses/bsd-license.php

[15]*Model*-view-controller (*MVC*) é um padrão de arquitetura de software que visa a separar a lógica de negócio da lógica de apresentação

[16]Mapeamento Objeto Relacional é uma técnica de desenvolvimento utilizada para reduzir a impedância da programação orientada aos objetos utilizando bancos de dados relacionais. As tabelas do banco de dados são representadas através de classes e os registros de cada tabela são representados como instâncias das classes correspondentes.

[17]Sistema Gerenciador de Banco de Dados.

> orientada a objetos, potencializa o reuso do código e facilita sua manutenção. (SANDERSON, 2009, p. 192)

O *Django* possui uma linguagem de Templates muito extensível e amigável. Através dela torna se possível separar a Interface do modelo de negócios.

A Figura 6.1 mostra o modelo comportamental do *Django*, nela está representada a interação entre os principais componentes do *framework.* As cores de cada componente representam a camada do modelo MVC a qual ele pertence.

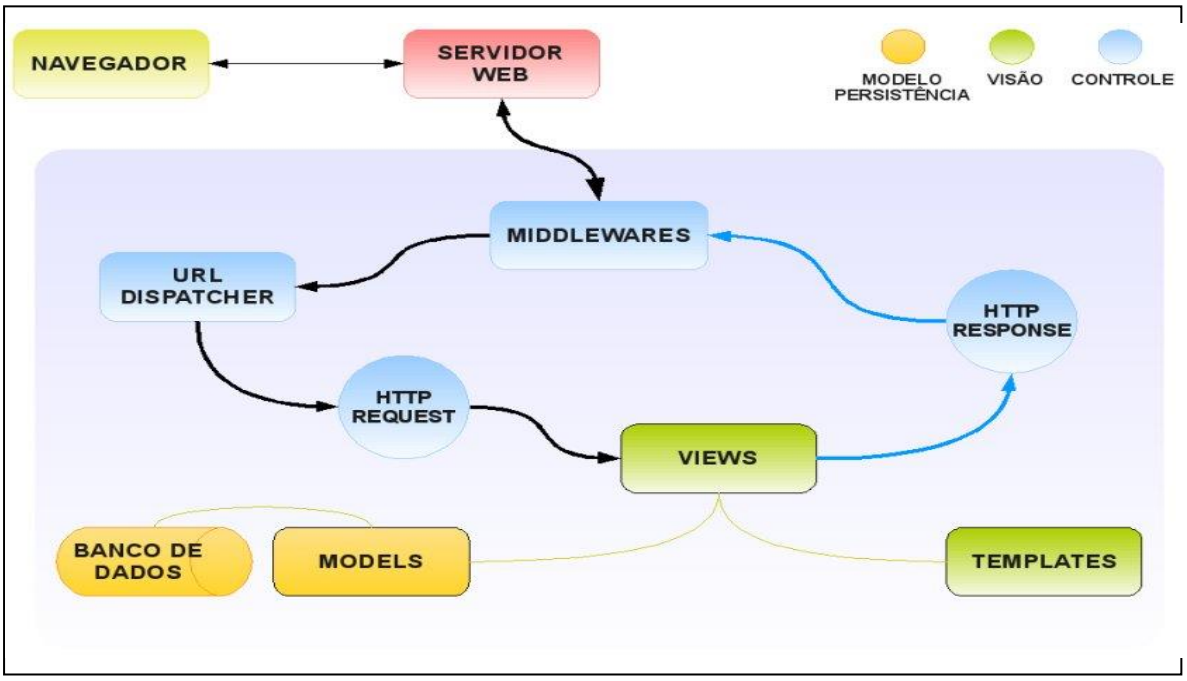


Figura 6.1: Arquitetura do Django Framework Fonte: (BISSEX,2008, p.192)

6.2 PERSISTÊNCIA DE DADOS

Nas aplicações *web* modernas, as ações geralmente envolvem interação com um banco de dados. Nos “bastidores” a aplicação conecta a um servidor

de banco de dados, recupera os dados necessários e os exibe em uma página web. De maneira análoga esse processo ocorre quando um usuário envia dados através do preenchimento de um formulário.

O *Django* é bem adequado para a criação de *webapps*[18] porquê ele tem embutidas muitas ferramentas fáceis e poderosas para a manipulação de banco de dados e realização de consultas.

O primeiro passo ao escrever um banco de dados de uma aplicação Web no *Django* é definir seus *Model*s essencialmente, o layout do banco de dados, com metadados adicionais.

Um *Model* é a única e definitiva fonte de dados sobre seus dados. Ele contém os campos essenciais e os comportamentos para os dados que você estiver armazenando.

O *Django* segue o *princípio DRY*. O objetivo é definir seu modelo de dados em um único local e automaticamente derivar coisas a partir dele.

Na Figura 6.2 temos um exemplo de modelo de Dados

```
django.db import models
class Poll(models.Model):
    question = models.CharField(max_length=200)
    pub_date = models.DateTimeField('date published')

class Choice(models.Model):
```

Figura 6.2 Modelo de Persistência Django

O Mapeador Objeto Relacional do *framework* irá traduzir esse código em um *script SQL* e então executá-lo no SGBD conFigurado na aplicação. Na Figura 6.3 é mostrado o *script* equivalente ao modelo mostrado anteriormente.

[18]Webapps - Aplicação *Web* é o termo utilizado para designar, de forma geral, sistemas de informática projetados para utilização através de um navegador, na internet ou em redes privadas (Intranet).

```
BEGIN;
CREATE TABLE "polls_poll" (
    "id" serial NOT NULL PRIMARY KEY,
    "question" varchar(200) NOT NULL,
    "pub_date" timestamp with time zone NOT NULL
);
CREATE TABLE "polls_choice" (
    "id" serial NOT NULL PRIMARY KEY,
    "poll_id" integer NOT NULL REFERENCES "polls_poll" ("id"),
    "choice" varchar(200) NOT NULL,
    "votes" integer NOT NULL
);
COMMIT;
```

Figura 6.3 Script SQL gerado pelo ORM Django

Todo o trabalho de criação de sincronização do modelo com o banco de dados é responsabilidade do ORM[19] do *Django*, o desenvolvedor deixa de trabalhar com tabelas relacionais e passa a trabalhar com classes do modelo de dados.

Na Figura 6.4 mostraremos um exemplo de CRUD[20] usando o modelo de persistência que foi criado anteriormente.

[19]ORM - Mapeamento Objeto Relacional

[20]*CRUD - (acrônimo de Create, Retrieve, Update e Delete) para as quatro operações básicas utilizadas em bancos de dados relacionais ou em interface para usuários para criação, consulta, atualização e destruição de dados.*

```
# Não há nenhuma enquete no sistema ainda.
>>> Poll.objects.all()
[]
# Crie uma nova enquete.
>>> import datetime
>>> p = Poll(question="What's up?", pub_date=datetime.datetime.now())
# Salve o objeto na base de dados. Você tem que chamar o save() explicitamente.
>>> p.save()
# Agora ele tem uma ID. >>> p.id
1
# Acesse colunas do banco de dados via atributos do Python.
>>> p.question
"What's up?"
>>> p.pub_date
datetime.datetime(2007, 7, 15, 12, 00, 53)
# Modifique os valores alterando os atributos e depois chamando o save()..
>>> p.pub_date = datetime.datetime(2007, 4, 1, 0, 0)
>>> p.save()
# objects.all() mostra todas as enquetes do banco de dados..
>>> Poll.objects.all()
[<Poll: Poll object>]
#Pegue a enquete cujo ano é 2007.
>>> Poll.objects.get(pub_date__year=2007)
<Poll: What's up?>
# Buscar por uma chave primária
>>> Poll.objects.get(pk=1)
<Poll: What's up?>
#Relacionamento 1 para n
>>> p = Poll.objects.get(pk=1)
>>> p.choice_set.create(choice='Not much', votes=0)
<Choice: Not much>
>>> p.choice_set.create(choice='The sky', votes=0)
<Choice: The sky>
>>> c = p.choice_set.create(choice='Just hacking again', votes=0)

# Objetos Choice possuem acesso via API aos seus objetos Poll
>>> c.poll
<Poll: What's up?>
# E vice-versa: Objetos Poll possuem acesso aos objetos Choice.
>>> p.choice_set.all()
[<Choice: Not much>, <Choice: The sky>, <Choice: Just hacking again>]
>>> p.choice_set.count()
3
#Apagando dado
>>> c = p.choice_set.filter(choice__startswith='Just hacking')
>>> c.delete()
```

Figura 6.4 exemplo crud usando modelos de persistência

6.3 VIEWS E TEMPLATES

A interface percebida pelo usuário é gerada através da camada de visão e dos *Templates* do *Django*. É nessa camada que ocorre a interação entre o usuário e o sistema.

O exemplo da Figura 6.5 mostra uma função da camada de visão que recebe uma requisição HTTP[21] e devolve como resposta uma página HTML[22] com o texto “Hora Atual:” e com o horário atual.

```
from django.http import HttpResponse
import time

def index(request):
   hora = time.asctime()
   return HttpResponse('<HTML><BODY> Hora atual: ',hora,'</BODY></HTML>')
```

Figura 6.5 Exemplo de view com HTML embutido Django (views.py)

Podemos notar que nesse exemplo simples, existem dois problemas graves que devem ser tratados.

- Primeiro existe código HTML embutido dentro de strings, o que dificulta a manutenção e induz erros, já que em projetos grandes existem os programadores chamados Front-End que são responsáveis pelo layout do sistema, e *Back-End* que são responsáveis pelas regras de negócio do sistema, e geralmente eles trabalham de forma colaborativa, porém separados.

[21] HTTP - Hypertext Transfer Protocol (do inglês, Protocolo de Transferência de Hipertexto) é um protocolo de comunicação (na camada de aplicação segundo o Modelo OSI) utilizado para sistemas de informação de hipermedia distribuídos e colaborativos.[1] Seu uso para a obtenção de recursos interligados levou ao estabelecimento da World Wide Web.

[22]HTML (acrônimo para a expressão inglesa HyperText Markup Language, que significa Linguagem de Marcação de Hipertexto) é uma linguagem de marcação utilizada para produzir páginas na Web.

- Qualquer mudança no design da página irá requerer mudança no código *Python*. E o *design* das páginas tendem a ser modificados com mais frequência do que o das regras de negócio.
- É mais eficiente se programadores puderem trabalhar no código *Python* e os designers no código *HTML* ao mesmo tempo. Ao invés de um ter de esperar pelo outro para editar o mesmo arquivo.

Para resolver esse problema, o *Django* oferece um conjunto de classes de *Templates.* Que tem a função de evitar o acoplamento das funcionalidades do modelo de visão no de controle.

Usando *Templates* ao invés de simplesmente inserir *HTML* em strings se resolvem todos esses problemas. Na Figura 6.6 temos um exemplo de código que faz a mesma coisa que o anterior, só que usando *Templates*.

```
import time
from django.http import HttpResponse

def index(request):
   horario = time.asctime()
   t = loader.get_template('polls/index.html')
   c = Context({'hora':horario})
   return HttpResponse(t.render(c))
```

Figura 6.6: View Hello World

```
<HTML>
        <BODY>
                Hora atual: {{hora}}
        </BODY>
<HTML>
```

Figura 6.7 Template HTML (hora.html)

Nesse exemplo notamos a clara separação entre o código da view e do código HTML que foi alocado no arquivo de Template. Com o uso de Templates

torna se possível a clara separação entre o trabalho do programador front-end para o back-end.

7. Desenvolvimento do Sistema de Postagens

Para possibilitar o teste de integração entre o framework e a plataforma da nuvem foi desenvolvido um sistema simples de postagem de comentário usando o framework *webapp* do *Google App Engine*.

Figura 7.1 Tela principal administração

7.1 LEVANTAMENTO DE REQUISITOS

A primeira atividade realizada no processo de modelagem foi o levantamento de requisitos, com a finalidade de identificar os requisitos funcionais e não-funcionais do sistema.

Nessa fase decidiu-se pela criação de uma aplicação simples para demostrar o processo de tradução de um aplicativo escrito originalmente para executar na plataforma do *Google App Engine* para uma aplicação que execute no framework *Django*.

Como requisito não funcional pode-se destacar:

- Maior simplicidade para que se permitir a explanação da tradução do sistema através do seu próprio código no trabalho de conclusão de curso.

Como requisitos funcionais temos:

- Criar, acessar e manipular modelos de dados
- Realizar requisições POST e GET
- Criar uma interface de administração para visualizar os dados do Banco de Dados.
- Promover criação e autenticação de usuários.

Todos os requisitos funcionais foram pensados com o fim de demostrar o processo de tradução de uma aplicação originalmente escrita para o framework *webapp* para executar no framework *Django*.

7.2 ANÁLISE DOS REQUISITOS

Após realização do levantamento de requisitos, é necessário conhecer o caminho mais objetivo para implementar as funcionalidades, além da identificação das entidades que deverão persistir no banco de dados relacional por meio do ORM e quais metainformações serão armazenadas sobre cada uma delas.

O administrador do sistema deverá ser capaz de inserir, acessar, modificar e excluir dados referentes as postagem e usuários cadastrados no sistema.

Ao usuário será provida a criação de um login e senha para acessar o sistema e a possibilidade de realizar postagens.

Abaixo temos a modelagem dos papeis de cada ator.

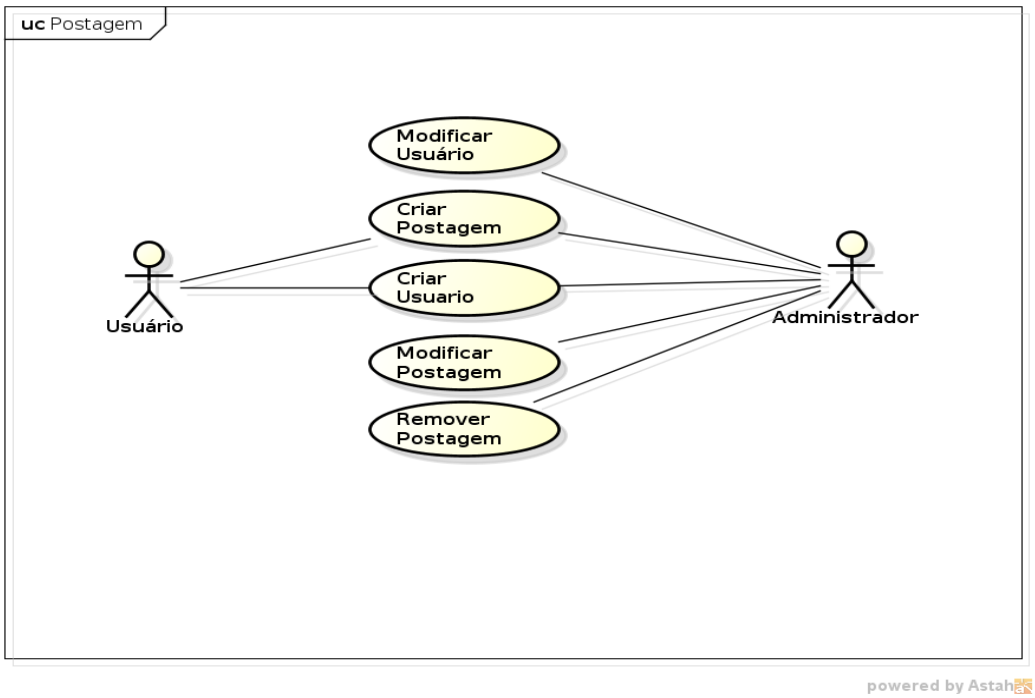


Figura 7.1: Caso de Uso

7.3 PERSISTÊNCIA DE DADOS

A modelagem do banco de dados foi feita através da ferramenta *DBDesigne*r na qual foram definidas as tabelas relacionais que iram representar os modelos de dados usados no sistema.

Optou-se pela modelagem de banco de dados relacionais (Veja Figura 7.2), já que o *Django* usa em sua essência essa abordagem.

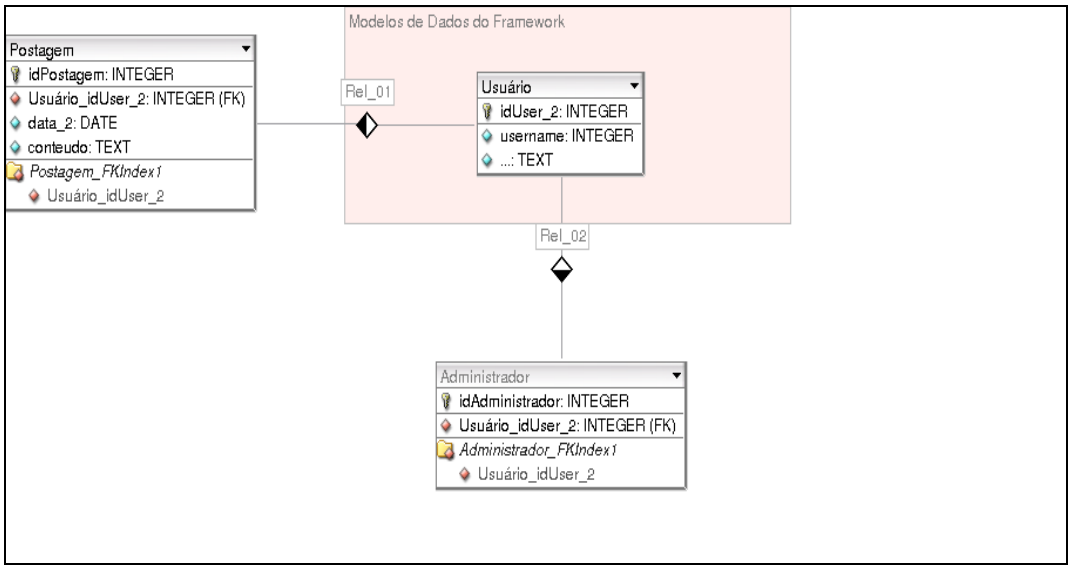


Figura 7.2: Modelagem modelos de dados

Na modelagem de dados do *webapp*, não é possível seguir o modelo descrito acima por usar um banco de dados não relacional. Por isso não se usam as chaves estrangeiras nem os relacionamento direto entre as tabelas. Ao invés disso foi usado o tipo de dados *UserProperty()* para representar o usuário.

```
class Postagem(db.Model):
    autor = db.UserProperty()
    conteudo = db.TextProperty()
    data= db.DateTimeProperty(auto_now_add=True)
```

Figura 7.3 Modelo de Dados *webapp*

No Apêndice C e D temos a Prototipação e as telas do sistema teste

8. INTEGRAÇAO

Existem vários métodos de integração entre o Framework *Django* e a plataforma do *Google App Engine*. Serão apresentadas alguns deles ressaltando as vantagens e desvantagens do uso de cada um.

8.1 USANDO A BIBLIOTECA *DJANGO* 0.96.1 PARA DESENVOLVER *TEMPLATES*

Código *HTML* embutido no código o torna bagunçado e difícil de manter. A melhor forma de separá-los é usar um sistema de *Template*, onde o *HTML* é mantido em arquivos separados, e usando uma sintaxe especial para indicar onde devem aparecer os dados da aplicação. A linguagem *Python* possui muitos sistemas de *templates*, entre eles *Django*, Quixote, *ClearSilver*, Cheetah, EZT. Graças a isso fica a critério de cada programador usar o motor de *Templates* de sua preferência para construir sua aplicação.

Por conveniência, o módulo *webapp* do *SDK* do *Google App Engine* tem embutido o módulo de *templates* das versões 0.96, o que facilita bastante a integração de sistemas *Django* mais simples e que possuam poucas rotinas de acesso ao banco de dados ou que não exijam mais que geração de páginas através de *templates*.

Através do código abaixo (veja Figura 8.1), torna-se hábil o uso do *Template Engine* do *Django.*

```
import os
from google.appengine.ext.webapp import template
```

Figura 8.1 **I**mportação da biblioteca *template*

Dessa forma modificou-se o código anterior para que se possa usar o *Template Engine* do *Django*. Observe a Figura 7.2.

```
class MainPage(webapp.RequestHandler):
        def get(self):
                guestbook_name=self.request.get('guestbook_name')
                postagens_query = Postagem.all().ancestor(
                guestbook_key(guestbook_name)).order('-data')
                postagens = postagens_query.fetch(10)
        if users.get_current_user():
                url = users.create_logout_url(self.request.uri)
                url_linktext = 'Logout'
        else:
                url = users.create_login_url(self.request.uri)
                url_linktext = 'Login'
        template_values = {
        'postagens': postagens,
        'url': url,
        'url_linktext': url_linktext,
        }
        path = os.path.join(os.path.dirname(__file__), 'index.html')
```

Figura 8.2 Arquivo *main.py*

Separadamente foi escrito o código HTML (veja o código na Figura 8.3).

```
<html>
  <body>
    {% for postagem in postagens %}
      {% if postagem.autor %}
        <b>{{ postagem.autor }}</b> wrote:
      {% else %}
        An anonymous person wrote:
      {% endif %}
      <blockquote>{{ postagem.conteudo | escape }}</blockquote>
    {% endfor %}

    <form action="/sign" method="post">
      <div><textarea name="conteudo" rows="3" cols="60"></textarea></div>
      <div><input type="submit" value="Enviar Postagem"></div>
    </form>

    <a href="{{ url }}">{{ url_linktext }}</a>

  </body>
</html>
```

Figura 8.3 Template html

O método *template.render* (*path, template_values*) receberá como argumento o caminho do *template HTML* e um dicionário de valores, e retornara uma página *HTML* com todas as variáveis e campos especiais expandidos como resposta.

8.1.1 Usando o padrão WSGI

O *Django* e o *Google App Angine* possuem suporte ao padrão *WSGI* para executar aplicações. Com isso, é possível usar quase toda a pilha do *Django* no *Google App Engine*. Para o desenvolvedor é necessário apenas ajustar e modificar os modelos de dados do *Django* para usar a API de armazenamento do *Google App Engine*.

Fez se *o upload* dos arquivos do framework para o servidor de desenvolvimento ou emulador e logo depois importá-lo usando um manipulador WSGI.

Outra mudança ao utilizar esse método é que ao invés de utilizar o manipulador de exceções *manage.py*, deve se tratar todas as exceções com o *Google App Engine*, sendo os *logs* acessíveis através do console de administração.

Na Figura 8.4 temos um exemplo de uso desse método de integração.

```
import logging, os
# Google App Engine imports.
from google.appengine.ext.webapp import util
# Force Django to reload its settings.
from django.conf import settings
settings._target = None
# Must set this env var before importing any part of Django
# 'project' is the name of the project created with django-admin.py
os.environ['DJANGO_SETTINGS_MODULE'] = 'project.settings'
import logging
import django.core.handlers.wsgi
import django.core.signals
import django.db
import django.dispatch.dispatcher
def log_exception(*args, **kwds):
        logging.exception('Exception in request:')
# Log errors.
django.dispatch.dispatcher.connect(
        log_exception, django.core.signals.got_request_exception)
# Unregister the rollback event handler.
django.dispatch.dispatcher.disconnect(
        django.db._rollback_on_exception,
        django.core.signals.got_request_exception)
def main():
        # Create a Django application for WSGI.
        application = django.core.handlers.wsgi.WSGIHandler()
        # Run the WSGI CGI handler with that application.
        util.run_wsgi_app(application)
if __name__ == '__main__':
        main()
```

Figura 8.4 Controller usado na integração usando WSGI

Juntamente deve se modificar o arquivo de configuração app.yaml para que roteie as requisições para a app correta através das requisições WSGI (Observe a Figura 8.5).

```
application: my_application
version: 1
runtime: python
api_version: 1
handlers:
- url: /static
        static_dir: static
- url: /.*
        script: main.py
```

Figura 8.5 ConFiguração do app.yaml para usar adaptador wsgi do django

Pode se também utilizar a versão de desenvolvimento do *Django*. O primeiro passo para isso é apagar as pastas *django/bin, django/contrib/admin, django/contrib/auth, django/contrib/databrowse, django/test* para reduzir o numero de arquivos .

O segundo passo será modificar o arquivo *main.py* da app original por uma versão que fornece através de homônimos bibliotecas fornecidas pela plataforma ao invés das bibliotecas fornecidas pelo próprio *Django*, incluindo sistema de armazenamento, geração de relatórios, funcionalidades do sistema, além de variáveis usadas pelo próprio *Django* durante o tempo de execução.Veja a Figura 8.6 abaixo.

```
# Google App Engine imports.
from google.appengine.ext.webapp import util
# Remove the standard version of Django.
for k in [k for k in sys.modules if k.startswith('django')]:
        del sys.modules[k]
# Force sys.path to have our own directory first, in case we want to import
# from it.
sys.path.insert(0, os.path.abspath(os.path.dirname(__file__)))
# Must set this env var *before* importing any part of Django
os.environ['DJANGO_SETTINGS_MODULE'] = 'settings'
import django.core.handlers.wsgi
import django.core.signals
import django.db
import django.dispatch.dispatcher
def log_exception(*args, **kwds):
        logging.exception('Exception in request:')
# Log errors.
django.dispatch.dispatcher.connect(
        log_exception, django.core.signals.got_request_exception)
# Unregister the rollback event handler.
django.dispatch.dispatcher.disconnect(
        django.db._rollback_on_exception,
        django.core.signals.got_request_exception)
def main():
        # Create a Django application for WSGI.
        application = django.core.handlers.wsgi.WSGIHandler()
        # Run the WSGI CGI handler with that application.
        util.run_wsgi_app(application)
if __name__ == '__main__':
        main()
```

Figura 8.6 arquivo main.py

Já que o *Google App Engine* não suporta os modelos de dados do *Django*, deixou-se em branco todas as variáveis de configuração de banco de dados do framework. Consequentemente o sistema de autenticação e o

sistema de administração do *Django* devem ser desabilitados já que dependem dos modelos de dados para funcionarem. O sistema de seções também depende dos modelos de dados e também devem ter seu uso evitado.

Por último deve se configurar o caminho para as pastas de *Templates* e arquivos estáticos no servidor (imagens, arquivos, folhas de estilo). Verifique a Figura 8.7.

```
import os
# 'project' refers to the name of the module created with django-admin.py
ROOT_URLCONF = 'project.urls'
MIDDLEWARE_CLASSES = (
   'django.middleware.common.CommonMiddleware',
#    'django.contrib.sessions.middleware.SessionMiddleware',
#    'django.contrib.auth.middleware.AuthenticationMiddleware',
   'django.middleware.doc.XViewMiddleware',
)
INSTALLED_APPS = (
#    'django.contrib.auth',
   'django.contrib.contenttypes',
#    'django.contrib.sessions',
   'django.contrib.sites',
)
ROOT_PATH = os.path.dirname(__file__)
TEMPLATE_DIRS = (
   # Put strings here, like "/home/html/django_templates" or
   # "C:/www/django/templates".  Always use forward slashes, even on Windows.
   # Don't forget to use absolute paths, not relative paths.
   ROOT_PATH + '/templates',
)
```

Figura 8.7 Settings.py

8.2 EXECUTANDO UM PROJETO DJANGO COMPLETO

O *Google App Engine* tem suporte nativo a alguns recursos do *Django*, como *Templates e Views*. Outras ferramentas que permitem o desenvolvimento de forma rápida como geração automática de formulários, interface de administração, e sistema de autenticação só iram funcionar fora da plataforma.

Existe uma *fork* não relacional do *Django* que permite a execução de projetos nativos do *framework Django* (incluindo Mapeamento Objeto Relacional) em banco de dados não relacionais, o qual é o sistema de banco de dados do *Google App Engine*. Esse *fork* se chama *Django-nonrel.*

Para executar o *Django* em um banco de dados não relacional, tudo o que se necessita é de suporte apropriado e o *Django-nonrel* fornece a união de várias ferramentas que permitem a execução de aplicativos nativo do *Django*.

O que significa que os usuários do *Django* poderão usar seu conhecimento para desenvolver no *Google App Engine* e que seus projetos irão funcionar sem qualquer modificação ou mudança de configuração. Além disso aplicações escritas para *Django-nonrel* podem ser portadas para para qualquer servidor que possua *Django* e que use *MongoDB* como Banco de Dados não relacional.

A seguir será demostrado um exemplo simples da conversão aplicação de teste escrita usando a biblioteca *webapp* para uma aplicação *Django-nonrel*.

A aplicação é composta por três arquivos, o primeiro *app.yaml (que é* mostrado na Figura 8.8) é responsável por armazenar as conFigurações da aplicação assim como fazer o roteamento das solicitações, o segundo arquivo *main.py* (Na Figura 8.9) é onde se encontram as regras de negócio da app e *index.html* (Figura 8.10) contêm a *template* que irá gerar a interface final com o usuário.

```
Application: Postagem
version: 1
runtime: python
api_version: 1
handlers:
- url: .*
  script: main.py
```

Figura 8.8 app.yaml

```
<html>
  <body>Ola
  {% if usuario %}
    {{ usuario.nickname }}!
    [<a href="{{ logout }}"><b>sign out</b></a>]
  {% else %}
    Mundo!
    [<a href="{{ login }}"><b>sign in</b></a>]
  {% endif %}
  <h2>Top 10 Most Recent Guestbook Entries</h2>
  {% for postagem in postagens %}
    <br>
    <small>[<i>{{ postagem.data.ctime }}</i>]</small>
    <b>
    {% if postagem.autor %}
      <code>{{ postagem.autor.nickname }}</code>
    {% else %}
      <i>anonymous</i>
    {% endif %}
    </b>
    wrote:
    {{ postagem.conteudo | escape }}
  {% endfor %}
  <hr>
  <form action="/sign" method=post>
    <textarea name=conteudo rows=3 cols=60></textarea>
    <br><input type=submit value="Enviar Postagem">
  </form>
  </body>
  </html>
```

Figura 8.9 *index.html*

```
import os

from google.appengine.api import memcache, users
from google.appengine.ext import db, webapp
from google.appengine.ext.webapp.template import render
from google.appengine.ext.webapp.util import run_wsgi_app
class Postagem(db.Model):
    autor = db.UserProperty()
    conteudo = db.TextProperty()
    data = db.DateTimeProperty(auto_now_add=True)
class MainHandler(webapp.RequestHandler):
    def get(self):
        usuario = users.get_current_user()
        postagens = memcache.get('postagens')
        if not postagens:
            postagens = Postagem.all().order('-data').fetch(10)
            memcache.add('postagens', postagens)
        context = {
            'usuario':      usuario,
            'postagens': postagens,
            'login':     users.create_login_url(self.request.uri),
            'logout':    users.create_logout_url(self.request.uri),
        }
        tmpl = os.path.join(os.path.dirname(__file__), 'index.html')
        self.response.out.write(render(tmpl, context))
class GuestBook(webapp.RequestHandler):
    def post(self):
        postagem = Postagem()
        postagem.conteudo = self.request.get('conteudo')
        postagem.put()
```

Figura 8.10 main.py

Para converter essa simples aplicação será dividido em 4 passos: convertê-la para a estrutura de arquivos do *Django*, converter os modelos de

dados, substituir os manipuladores de requisição, substituir as conFigurações de roteamento de URL.

8.2.1 Estrutura de arquivos do DJANGO

O primeiro passo é entender que uma aplicação *Django* é feito para ser parte de um ecossistema de desenvolvimento maior do qual se encontra os projetos *Django*. *Django* possui o conceito de projeto, que é essencialmente um conjunto de conFigurações globais que controlam o comportamento de uma ou mais aplicações. Em analogia pode se pensar no projeto como um um site que possui uma ou mais aplicações que executaram por baixo como blogs, sistema de autenticação, fóruns, formulário para contatos entre outros.

Um projeto *Django* contem basicamente 4 arquivos(__init__.py, settings.py, manage.py, urls.py) que podem ser acompanhados de um ou mais subdiretórios. Abaixo temos a descrição de cada um desses arquivos:

- manage.py → interface via linha de comandos das aplicações.
- urls.py → roteamento de urls do projeto
- settings.py → configuração do projeto
- __init__.py → informa ao interpretador *Python* que o diretório contém um pacote

Como dito antes, um projeto contém uma ou mais aplicações. Cada aplicação fica em subdiretórios dentro do projeto principal e neles existem quatro arquivos principais (__init__.py, models.py, views.py, tests.py) e opcionalmente um diretório para armazenar templates. Abaixo a função de cada um dos principais arquivos da aplicação:

- views.py → manipulador de requisições
- models.py → armazena os modelos de dados
- tests.py → onde são realizados os testes de unidade
- __init__.py → informa que esse diretório é um pacote *Python*

O próprio framework *Django* fornece ferramentas que facilitam a criação da estrutura de arquivos através da interface via linha de comandos.

8.2.2 MODELOS DE DADOS.

Alguns passos foram necessários para portar uma aplicação com modelos de dados do App Engine para o *Django*:

1. Mover o conteúdo dos modelos de dados para o arquivo models.py do projeto.
2. Importar os modelos de *django.db.models.Model* ao invés de *google.appengine.ext.db.Model*
3. Substituir os tipos de dados do *App Engine* por correspondentes no *Django*.

   Foi usado o Modelo de dados do *App Engine* definido pela classe Postagem:

```
from google.appengine.ext import db
class Postagem(db.Model):
    autor = db.UserProperty()
    conteudo = db.TextProperty()
    data = db.DateTimeProperty(auto_now_add=True)
```

Figura 8.11 : Modelo de Dados App Engine - Classe Postagem

```
>>> from guestbook.models import Postagem
>>> Postagem.objects.count()
0
>>> postagem = Postagem(conteudo='Hi!')
>>> postagem.save()
>>> Postagem.objects.count()
1
>>> postagem = Postagem.objects.all()[0]
>>> postagem.conteudo
'Hi!'
```

Figura 8.12 Testando o Modelo Postagem

```
from django.db import models
from django.contrib.auth.models import User
class Postagem(models.Model):
    autor = models.ForeignKey(User, null=True, blank=True)
    conteudo = models.TextField()
    data = models.DateTimeField(auto_now_add=True)
```

Figura 8.13 Modelo de Dados Django - Postagem

Seguindo os três passos o código ficará assim:

Substitui-se o tipo de dados do campo autor por um tipo de dados que simula uma chave estrangeira que faz referência à classe User do *Django* e logo depois adicionando as propriedades *null=True, blank= True*, que indica que aquele campo pode ser salvo com valores vazios ou nulos.

O campo conteúdo que usa o modelo de dados *db.TextProperty* é substituído por *models.TextField()* e o campo data substitui *db.DateTimeProperty* por *models.DateTimeField* mantendo a propriedade

auto_now_add=True, para que no momento da gravação no banco de dados seja registrado automaticamente a data e o horário da saudação.

Podemos testar via linha de comandos os tipos de dados que foram criados através do comando *python manage.py shell.*

Depois de testado e funcionando pode se escrever rotinas de teste de unidade mesmo antes de escrever a aplicação. Para isso foi usado o arquivo tests.py da aplicação.

```
from django.test import TestCase
from guestbook.models import Postagem
class SimpleTest(TestCase):
    def setUp(self):
        Postagem(conteudo='This is a test postagem').save()
    def test_setup(self):
        self.assertEqual(1, len(Postagem.objects.all()))
        self.assertEqual('This is a test postagem', Postagem.objects.all()[0].conteudo)
```

Figura 8.14 Teste de unidade

8.2.3 Manipulador de requisição

Os manipuladores de requisição do *Django* são funções *Python* que substituem todas as funcionalidades dos manipuladores do *Google App Engine* e por isso mesmo devem ser substituídas seguindo alguns passos:

- Substituir os manipuladores por views no arquivo views.py
- Substituir todas as consultas ao banco do App Engine por consultas do *Django*.
- Enviar as respostas das solicitações para algum template.

8.2.4 Manipulador GET

Foi convertido as *views* de cada manipulador de solicitação da aplicação. Iniciando com o *metodo get()* da classe *MainHandler.*

```
import os
from google.appengine.api import memcache, users
from google.appengine.ext import webapp
from google.appengine.ext.webapp.template import render
class MainHandler(webapp.RequestHandler):
   def get(self):
      usuario = users.get_current_user()
      postagens = memcache.get('postagens')
      if not postagens:
         postagens = Postagem.all().order('-data').fetch(10)
         memcache.add('postagens', postagens)
      context = {
         'usuario':     usuario,
         'postagens': postagens,
         'login':    users.create_login_url(self.request.uri),
         'logout':   users.create_logout_url(self.request.uri),
      }
      tmpl = os.path.join(os.path.dirname(__file__), 'index.html')
      self.response.out.write(render(tmpl, context))
```

Figura 8.15 Classe *MainHandler* no *Google App Engine*

```
from django.core.cache import cache
from django.views.generic.simple import direct_to_template
from guestbook.forms import CriarFormularioPostagem
from guestbook.models import Postagem
def list_postagens(request):
    postagens = cache.get(MEMCACHE_GREETINGS)
    if postagens is None:
        postagens = Postagem.objects.all().order_by('-data')[:10]
        cache.add(MEMCACHE_GREETINGS, postagens)
    return direct_to_template(request, 'guestbook/index.html',
        {'postagens': postagens, 'form': CriarFormularioPostagem()})
```

Figura 8.16 View Correspondente à classe MainHandler no Django

Ao invés de usar a *API memcache* do *Google App Engine* optou-se pelo uso pelo modulo de cache do *Django*, com isso a aplicação se torna independente da plataforma.

Os dicionários de dados também são renderizados com o template usando a view genérica *direct_to_template* que recebe como argumentos uma *request,* o endereço do template e o dicionário de dados. Nessa fase se optou pelo uso do módulo de geração de formulários fornecidos pelo *framework Django* passando *CriarFormularioPostagem* como elemento do dicionário de dados para a renderização do *template*.

Também foi modificada a forma como é realizada a consulta ao banco de dados por todos os objetos Postagem já que agora serão usados os modelos da camada do *Django* e não mais diretamente o banco do *Google App Engine*.

```
class Postagem(db.Model):
    autor = db.UserProperty()
    conteudo = db.TextProperty()
    data = db.DateTimeProperty(auto_now_add=True)

postagens = Postagem.all().order('-data').fetch(10)
```

Figura 8.17 Busca por todas as Saudaçoes do Banco no App Engine

```
from guestbook.models import Postagem
...
postagens = Postagem.objects.all().order_by('-data')[:10]
```

Figura 8.18 Busca por todas as Saudaçoes no Banco de Dados do Django

Nota se nesse exemplo uma grande vantagem no uso do modelo MVC aplicado pelo *framework Django* já que há uma clara separação entre a declaração dos modelos de dados e a camada da *view*, o que não acontece no *Google App Engine*.

Uma outra vantagem que se nota ao usar o *Django* é o princípio *DRY(don't repeat yourself),* que reúne padrões de rotinas e cria vários atalhos para simplificar tarefas repetitivas como a renderização de templates.

Para isso *Django* possui o conceito de *generic views* que ao invés de apenas mapear os resultados e enviá-los para o *template*, simplifica o processo de desenvolvimento abstraindo rotinas recorrentes.

Neste caso a view genérica *direct_to_template* foi usada para de modo a renderizar o formulário gerado automaticamente pelo módulo *Forms* e os objetos Postagem recuperados do cache de memória ou do banco de dados

dependendo do estado do sistema. A *view* genérica pode também enviar automaticamente para o *template* variáveis extras como o objeto *request* e o usuário atual.

8.2.5 TEMPLATES

O próximo passo é modificar o *template* do App Engine para que seja renderizado pelo *Django*.

```
{% for postagem in postagens %}
   <br>
   <small>[<i>{{ postagem.data.ctime }}</i>]</small>
   <b>
   {% if postagem.autor %}
      <code>{{ postagem.autor.nickname }}</code>
   {% else %}
      <i>anonymous</i>
   {% endif %}
   </b>
   wrote:
   {{ postagem.conteudo | escape }}
{% endfor %}
```

Figura 8.19 Template do Google App Engine

```
{% for postagem in postagens %}
   <br>
   <small>[<i>{{ postagem.data.ctime }}</i>]</small>
   <b>
   {% if postagem.autor %}
      <code>{{ postagem.autor.username }}</code>
   {% else %}
      <i>anonymous</i>
   {% endif %}
   </b>
   wrote:
   {{ postagem.conteudo }}
{% endfor %}
```

Figura 8.20 *Template* Traduzido para ser renderizado pelo *Django*

Apenas duas mudanças foram necessárias, a primeira é o atributo autor.*nickname* já que o tipo de dados *User* do *Django* possui o atributo username. A segunda mudança foi a retirada do filtro { |*escape* } que têm a função de converter automaticamente os caracteres especiais para a sintaxe das strings do *HTML*, já que as versões mais novas do *Django*, já realizam a conversão automática no momento da renderização dos *templates.*

Na Figura 8.21 temos um exemplo do uso do { |*escape* }:

```
< é convetido para &lt;
> é convertido para &gt;
' (aspas simples) é convertido para&amp;#39;
" (aspas duplas) é convetido para &amp;quot;
& é convertido para &amp;amp;
```

Figura 8.21 Uso do |escape

A semelhança dos templates não ocorre por acaso, já que o próprio *Google App Engine* utiliza a base da engine de renderização de templates do *Django* 0.96. Sendo necessário apenas modificar os tipos os atributos e os tipos de dados, e remover o filtro de conteúdo desnecessário.

8.2.6 Autenticação

Para portar a aplicação também deve se modificar o modo como o usuário irá se autenticar no sistema. Existem diversos modelos de autenticação tanto no *Google App Engine* quanto no *Django*.

No *Google App Engine* pode se escolher fazer um sistema sem autenticação, com autenticação através de uma conta do Google, através de um domínio Google Apps, ou através do sistema de identificação do OpenID.

O *Django* permite o desenvolvimento de um sistema sem autenticação, usando o módulo de autenticação do *Django* ou atraves do sistema de identificação OpenID através do módulo django-socialauth.

ConFigurar o sistema de autenticação no *Django* é um pouco mais complexo do que usando *webapp* já que é necessário desenvolver também um template para o login, logout e para o gerenciamento dos usuários e no *webapp* o tudo isso fica a cargo do Google.

Na Figura 8.22 veremos o trecho de autenticação do *webapp* e o código portado para *Django*.

```
Ola
{% if usuario %}
   {{ usuario.nickname }}!
   [<a href="{{ logout }}"><b>sign out</b></a>]
{% else %}
   Mundo!
   [<a href="{{ login }}"><b>sign in</b></a>]
{% endif %}
```

Figura 8.22

```
Ola
{% if usuario.is_authenticated %}
   {{ usuario.username }}!
   [<a href="{% url django.contrib.auth.views.logout %}"><b>sign out</b></a>]
{% else %}
   Mundo!
   [<a href="{% url django.contrib.auth.views.login %}"><b>sign in</b></a>]
{% endif %}
```

Figura 8.23

Ao invés de usar a URL passada através das variáveis logout e login pelo dicionários de contexto pode se usar o recursos da tag *{%url django.contrib.auth.views.logout %}* que realiza uma busca através da tabela de roteamento(arquivo urls.py) pela view *django.contrib.auth.view.logout* e gera uma URL absoluta que irá direcionar o usuário para a página de logout.

8.3 MANIPULADOR POST

Usando *webapp* manipulação do formulário é feita através de um manipulador *POST* que recebe um dicionário constituído por rótulos e dados dos campos do formulário.

Abaixo mostraremos o trecho original e a versão portada para o *Django*:

```
class Guestbook(webapp.RequestHandler):
    def post(self):
        postagem = Postagem()
        postagem.conteudo = self.request.get('conteudo')
        postagem.put()
        memcache.delete('postagens')
        self.redirect('/')
```

Figura 8.24 Manipulador *POST* no webapp

```
from django.http import HttpResponseRedirect
from guestbook.forms import CriarFormularioPostagem

def create_postagem(request):
    if request.method == 'POST':
        form = CriarFormularioPostagem(request.POST)
        if form.is_valid():
            postagem = form.save(commit=False)
            if request.usuario.is_authenticated():
                postagem.autor = request.usuario
            postagem.save()
            cache.delete('postagens')
    return HttpResponseRedirect('/guestbook/')
```

Figura 8.25 Manipulador POST no DJANGO

Ao contrário do *webapp* o *Django* não faz o tratamento direto da requisição, mas sim delega essa função à função CriarFormularioPostagem e que retorna um objeto da classe forms que cuida da validação e tratamento das

entradas do formulário e que após salvo retorna um objeto da classe de modelo de dados Postagem que pode então ser salva no banco de dados. Demonstrando a importância do uso da filosofia DRY(don't repeat yourself) que permite o tratamento de formulários em várias partes da aplicação sem a necessidade de reescrever o código.

Outro ponto importante a se notar é a separação do que é visão do que é manipulação de banco de Dados, já que o tratamento e geração do objeto é delegado à uma função externa especializada nisso. Reforçando a importância do uso do padrão MVC para promover o reuso e aumentar a legibilidade do código.

Como foi necessário adicionar dados do usuário que realizou a postagem após a validação dos dados do formulário fez se a chamada à função *form.save()* passando o atributo *commit=false* para desativar o salvamento automático, adicionou se então os dados do usuário ao objeto de modelo e fez-se o salvamento explicito através da chamada postagens.save().

8.4 FORMULÁRIOS

A maior parte do código escrito em uma aplicação web trata da criação, apresentação, envio e validação de formulários. *Django* fornece um módulo que trata especificamente disso.

Django forms é uma abstração de formulários HTML usado para apresentar, capturar e validar dados. Os objetos da classe Forms substitui as práticas comumente usadas pelos desenvolvedores para gerenciar o mapeamento de campos de formulários HTML, parâmetros de requisições POST e validação encapsulando estas funcionalidades dentro de objetos configuráveis e testáveis.

Esses objetos podem ser instanciados de duas formas associados ou não associados. Se o objeto instanciado não possuir parâmetros ele será não associado e simplesmente irá gerar o formulário HTML após renderizado, mas se o formulário for postado, ele se tornará associado já que receberá como argumento o dicionário de dados da requisição POST.

Abaixo será mostrado um exemplo da criação de uma classe Form a partir de um Modelo de Dados.

```
#Modelo de Dados
class MeuModelo(models.Model):
    versao= models.CharField(max_length=32)
    conteudo = models.CharField(max_length=256)
    data = models.DateField(blank=True, null=True)

# Classe Formulário
class MeuFormulario(forms.Form):
    versao = forms.CharField(max_length=32)
    conteudo = forms.CharField(max_length=256)
    data = forms.DateField(required=False)
```

Figura 8.26 Classe Formulário

Como o Framework prega a filosofia *DRY*, pode se simplificar a *Classe Form* para usando a derivação *ModelForm* como demonstrado a seguir:

```
# model-form class
class MeuFormulario(forms.ModelForm):
    class Meta:
        model = MeuModelo
```

Figura 8.27 Classe Formulário

Dessa forma é gerada uma classe associada ao modelo de dados e que irá gerar automaticamente todos os campos do formulário com os mesmos rótulos e com os mesmos tipos de dados, e que servirá também para tornar os dados persistentes após a sua validação.

Voltando à aplicação teste, mostraremos como ficou a classe *Forms* dessa aplicação baseado no Modelo de Dados

```
from google.appengine.ext import db
class Greeting(db.Model):
    author = db.UserProperty()
    content = db.TextProperty()
    date = db.DateTimeProperty(auto_now_add=True)
```

Figura 8.28 Modelo de dados

```
from django import forms
from guestbook.models import Postagem
class CriarFormularioPostagem(forms.ModelForm):
    class Meta:
        model = Postagem
        exclude = ('autor', 'data')
```

Figura 8.29 Classe Formulário

Verificamos grande semelhança com a classe *MeuFormulario* mostrada no exemplo acima com a diferença que que foi acrescentada a variável exclude à metaclasse para que os atributos autor e data não gerem campos de entrada de dados no formulário *HTML* nem sejam validados e gravados automaticamente no banco de dados, já que o atributo data é obtido automaticamente através do relógio do sistema, e o atributo autor é fornecido através do sistema de autenticação e obtido através da solicitação *POST*.

Por mais complexo que seja o modelo ao usar model-forms django automaticamente cria um formulário para ele. Com isso podemos usar o model-form diretamente no template de forma que o arquivo original fica da seguinte forma:

```
<form action="/sign" method="post">
  <div><textarea name="conteudo" rows="3" cols="60"></textarea></div>
  <div><input type="submit" value="Enviar Postagem"></div>
 </form>
```

Figura 8.30 Formulário *HTML* usando *webapp*

```
<form action="/guestbook/sign" method="post">{% csrf_token %}
      <table>{{ form }}</table>
      <input type="submit" value="Enviar Postagem" />
 </form>
```

Figura 8.31 Formulário *HTML* usando *Django model-forms*

8.5 ROTEAMENTO DE URLS

Tanto o *Django* quanto o *webapp* funcionam com um sistema de roteamento de urls para encaminhar as solicitações *GET* para as *views* correspondetes. No *Google App Engine* as rotinas de roteamento são definidas através do arquivo *app.yaml*, já no *Django* é de responsabilidade do arquivo *settings.py* que por sua vez delega a tarefa para o *urls.py*.

Por conta dos diferentes *templates* para criar e mostrar as postagens o arquivo *urls.py* possui dois padrões de *URL* em contrapartida o arquivo *app.yaml* do *webapp* apenas uma como mostrado a seguir.

```
# Arquivo app.yaml do webapp
- url: /.*
  script: main.py

# Arquivo urls.py do Django
from django.conf.urls.defaults import *
urlpatterns = patterns('guestbook.views',
    (r'^$',     'list_greetings'),
    (r'^sign$', 'create_greeting'),
)
```

Figura 8.32 Tradução das ConFigurações de Roteamento de *URL*

A última coisa a se fazer antes executar o sistema teste é modificar o arquivo de configuração do projeto *settings.py* a fim de permitir a execução da aplicação Postagem alem disso foram habilitados os módulos *django.contrib.sessions* (gerenciamento de sessões) e django.contrib.auth (sistema de autenticação).

```
from djangoappengine.settings_base import *
import os
SECRET_KEY = '=r-$b*8hgw3sc58&9t0twan5ch1k-3d3vfc4(wk0rn3wa1dhvi'
INSTALLED_APPS = (
    'djangoappengine',
    'djangotoolbox',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'postagem',
)
MIDDLEWARE_CLASSES = (
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
)
TEMPLATE_CONTEXT_PROCESSORS = (
    'django.contrib.auth.context_processors.auth',
    'django.core.context_processors.request',
)
LOGIN_REDIRECT_URL = '/postagem/'
ADMIN_MEDIA_PREFIX = '/media/admin/'
MEDIA_ROOT = os.path.join(os.path.dirname(__file__), 'media')
TEMPLATE_DIRS = (os.path.join(os.path.dirname(__file__), 'templates'),)
ROOT_URLCONF = 'urls'
```

Figura 8.33 Arquivo de ConFiguração do Projeto *settings.py*

8.6 EXECUTANDO E IMPLANTANDO O PROJETO NA NUVEM

Para testar o projeto *Django* localmente executou-se:

$ python manage.py runserver

Deste modo o projeto pode ser acessado através do endereço http://localhost:8000 por meio de um navegador web.

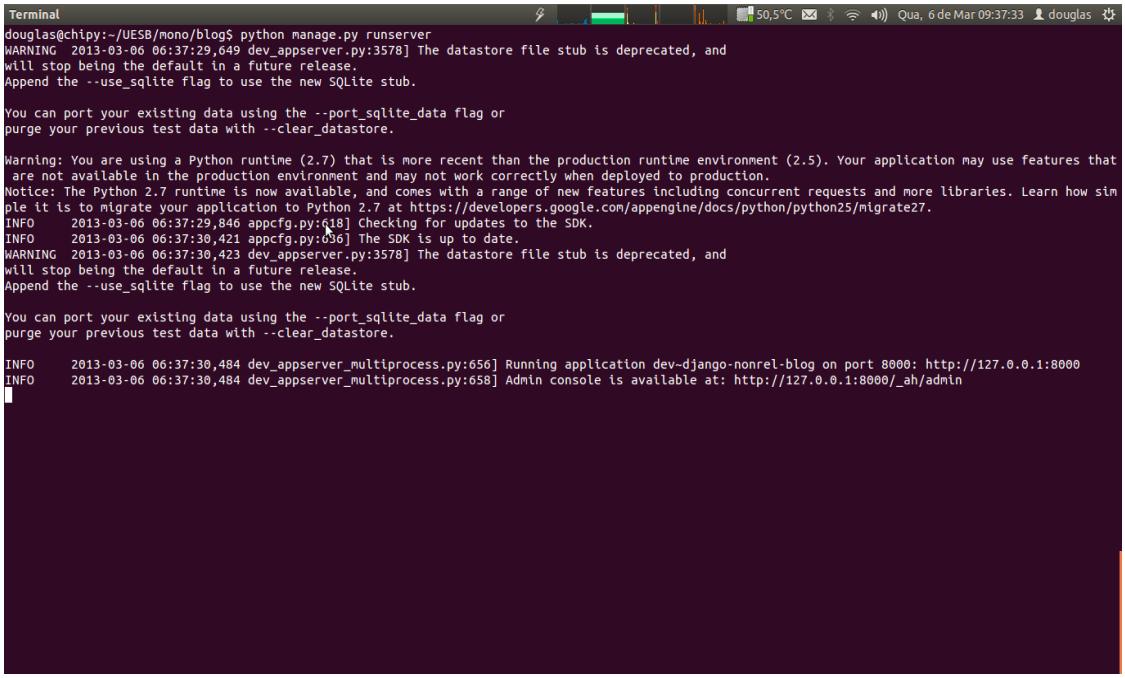


Figura 8.34 Executando o servidor de desenvolvimento

Para enviar o projeto para a nuvem deve-se primeiro criar uma aplicação através da interface de administração do GAE através do endereço http://appengine.google.com e configurar o arquivo *app.yaml* com o mesmo nome da aplicação criada.

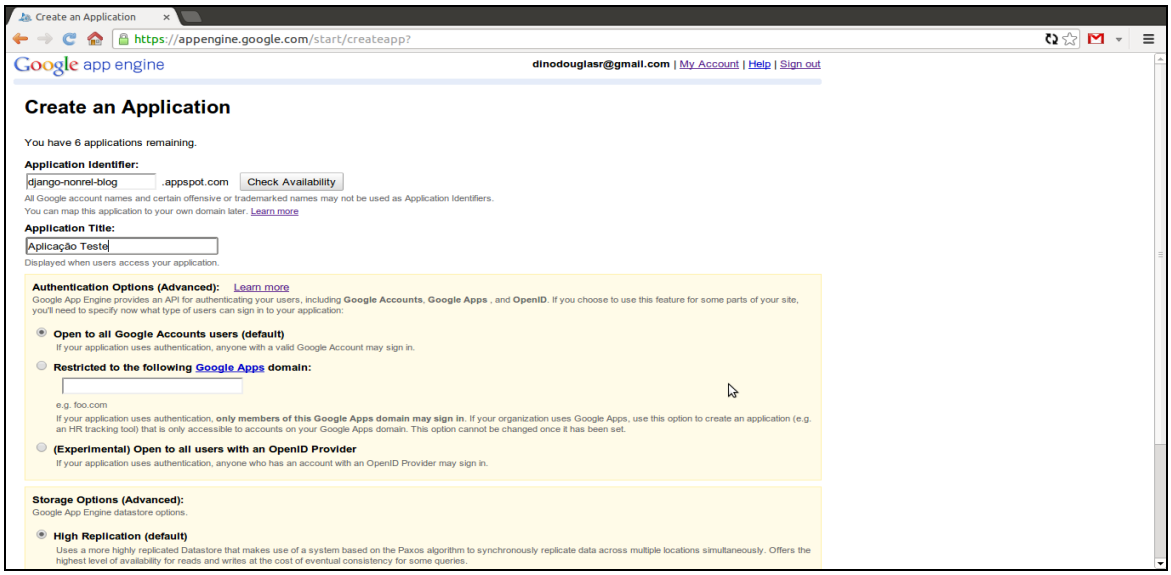


Figura 8.35 Criando uma aplicação no *Google App Engine*

Depois disso executou-se.

$python manage.py syncdb

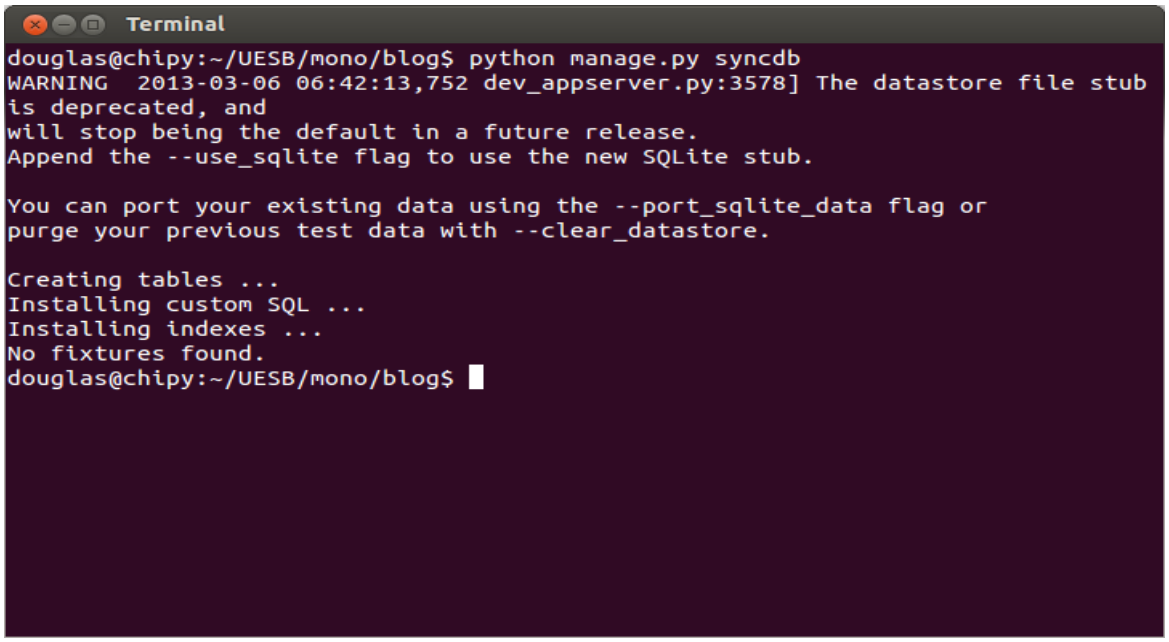


Figura 8.36 Geração do banco de Dados

Para que fossem geradas todas as tabelas relativas aos modelos de dados declarados. E finalmente:

$python manage.py deploy

Esse último comando realizou a implantação do sistema desenvolvido na Nuvem da plataforma do *Google App Engine*.

O Resultado é mostrado na figura 8.37

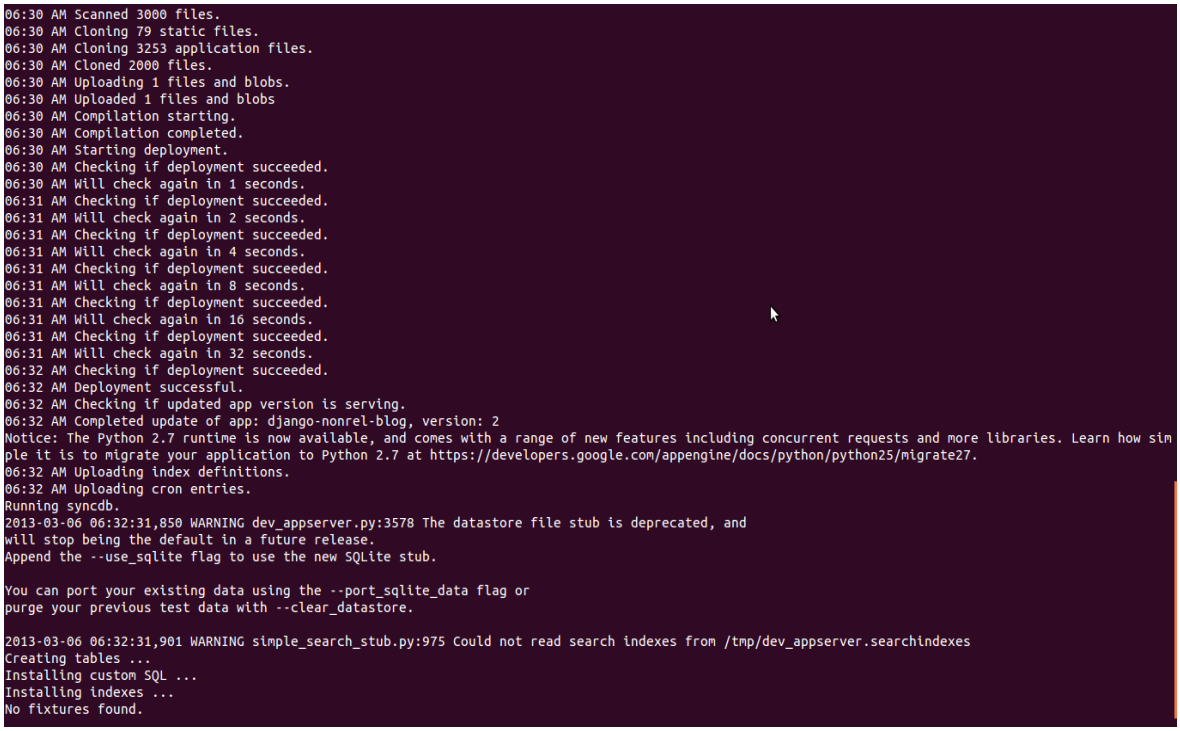


Figura 8.37 Realizando a Implantação do Sistema na Plataforma de Nuvem

Abaixo na figura 8.38 pode-se ver a página inicial do sistema já implantado.

Figura 8.38 – Pagina inicial do sistema implantado

9. CONCLUSÃO

Uma das vantagens de se usar o *Django-nonrel* é que se por algum motivo for decidido pela retirada do sistema do *Google App Engine* para um serviço de *hosting* tradicional, a única coisa a se fazer será a modificação das conFigurações no arquivo settings.py para que se seja utilizado outro banco *NoSQL* (Não relacional) disponível no mercado e compatível com *Django*, como é o caso do *MongoDB* e após isso remover o modulo *djangoappengine* do projeto. Ou seja o sistema se torna portável com poucas operações e com nenhuma mudança no código fonte para que isso aconteça.

Com isso o desenvolvedor não ficará com seu sistema preso a uma única plataforma de desenvolvimento que a qualquer momento pode modificar as regras de fornecimento de seus serviços, como prevê seus termos de serviço.

Muitas empresas se preocupam em evitar *vendor lock in* que é quando uma empresa confia as suas aplicações e dados nas mãos de um único fornecedor o que dificulta ou impossibilita a migração dos dados e do próprio sistema para uma solução de outro fornecedor. Usando o *Django-nonrel* é possível aproveitar todas as vantagens da plataforma do *Google App Engine* (escalabilidade, disponibilidade, poder de processamento entre outros) sem se preocupar em futuramente ficar preso à um único fornecedor.

Antes do *Django-nonrel*, haviam dois projetos que realizavam a integração parcial do Framework à plataforma de desenvolvimento em nuvem do *Google App Engine*. Um dos projetos era o *app-engine-patch* que ainda hoje é bem popular, o outro foi o *Google App Engine Helper* for *Django*. Esses dois projetos se tornaram obsoletos pois exigiam que o programador utilizassem os modelos de dados personalizados ao invés do sistema ORM do próprio *Django*, com isso se impossibilitou o uso de todos os módulos do *Django* que usam o banco de dados(Autenticação, *ModelForms, ModelAdmin*, Sessão entre outros) além de ter obrigado ao desenvolvedor fazer modificações em todas as chamadas do sistemas que fizessem referência a algum modelo de dados.

Com o *Django-nonrel*, pode se usar os modelos de dados nativos do *Django* de forma direta e transparente. Isso permite a reutilização de aplicativos escritos originalmente em *Django* sejam executados no *sandbox* de plataformas que suportem a linguagem *python*.

As abordagens anteriores exigiam modificação de parte significativa da aplicação, em alguns casos obrigava o uso de classes de modelo especiais do Google (o que prendia a aplicação à plataforma), e não permitia o uso de aplicativos que já haviam sido escritos para *Django*. Após a criação do projeto *Django-nonrel* resolveram se esses problemas e esses dois projetos se tornaram obsoleto e foram descontinuados e todos os esforços da comunidade de desenvolvedores se voltaram para o *Django-nonrel*.

Como trabalho futuro deve-se desenvolver uma ferramenta para a automatização da conversão de aplicativos escritos originalmente usando *webapp* para *Django-nonrel.*

REFERÊNCIAS

ARMBRUST, M., FOX, A., GRIFFITH, R., JOSEPH, A. D., KATZ, R., KONWINSKI, A., LEE, G., PATTERSON, D., RABKIN, A. & STOICA, I. 2010. **A view of cloud computing**. Communications of the ACM, 53, 50-58.

BUYYA, R., Yeo, C. S., Venugopal, S., Broberg, J., and Brandic,. **Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing** the 5th utility. Future Gener. Comput. Syst., 2009

CARR, Nicholas G., 2008. **The big switch: Rewiring the world, from Edison to Google**. New York: Norton, p. 142.

CEARLEY, D. et al – **Hype Cycle for Applications Development – Gartner Group Reporter number G00147982**. Disponível em: <http://www.gartner.com>. Acessado em: 08 mai 2011.

MARINOS, Alexandros; BRISCOE, Gerard. Community Cloud Computing. In: 4TH IEEE INTERNATIONAL CONFERENCE ON CLOUD COMPUTING TECHNOLOGY AND SCIENCE, ., 2012, Taipei. **Cloud Computing Technology and Science.** London: Fsl, 2009. v. 1, p. 1 - 15.

NIST. The **NIST Definition of Cloud Computing**. Disponível em: <http://csrc.nist.gov/publications/drafts/800-145/Draft-SP-800-145_cloud-definition.pdf>. Acessado em: 08 mai 2011.

RODRIGUEZ, M. A. and Neubauer, P. (2010). **Constructions from dots and lines**. Bulletin of the American Society for Information Science and Technology, 36(6):35–41

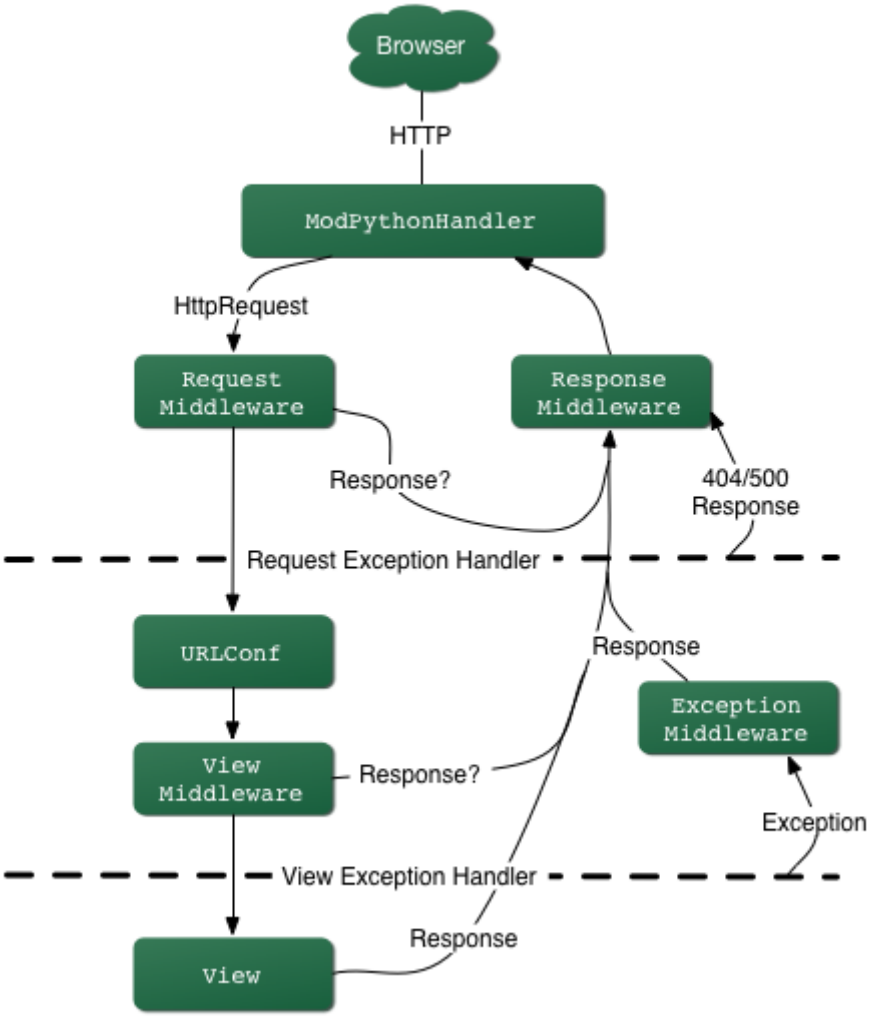
SANDERSON, Dan. **Programing *Google App Engine*:** Build and run scalable web apps on Google Infraestruture. 2. ed. Sebastopol: O Relly, 2009. 367 p

SILVA, Fabrício Rodrigues Henriques da.**UM ESTUDO SOBRE OS BENEFÍCIOS E OS RISCOS DE SEGURANÇA NA UTILIZAÇÃO DE CLOUD COMPUTING.**2009. 22 f. Monografia (3º) - Curso de Ciência da Computação, Unisuan, Jacarepaguá, 2009.

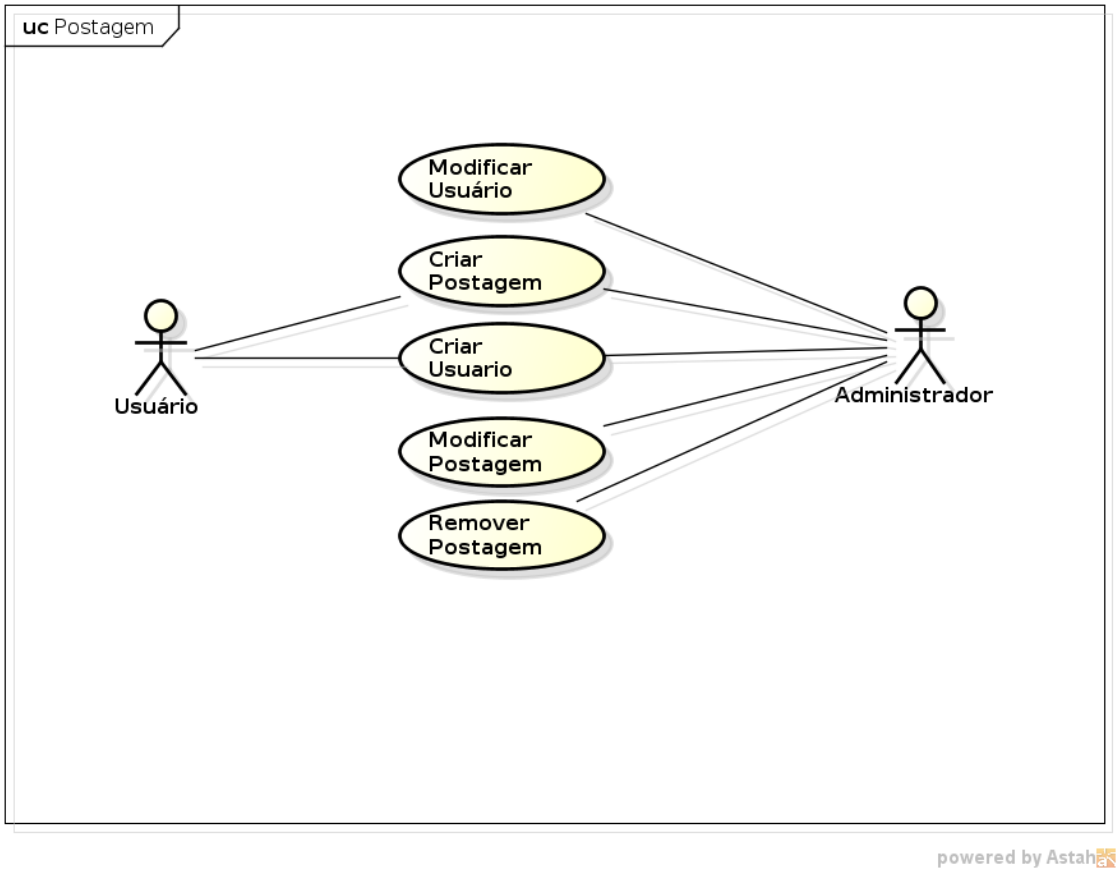
TAURION, C**. Computação em Nuvem: Transformando o mundo da tecnologia da informação**. Rio de Janeiro: Brasport, 2009.

VAQUERO, L. M., RODERO-MERINO, L., CACERES, J. & LINDNER, M. 2008. **A break in the clouds: towards a cloud definition**. ACM SIGCOMM Computer Communication Review, 39, 50-55.

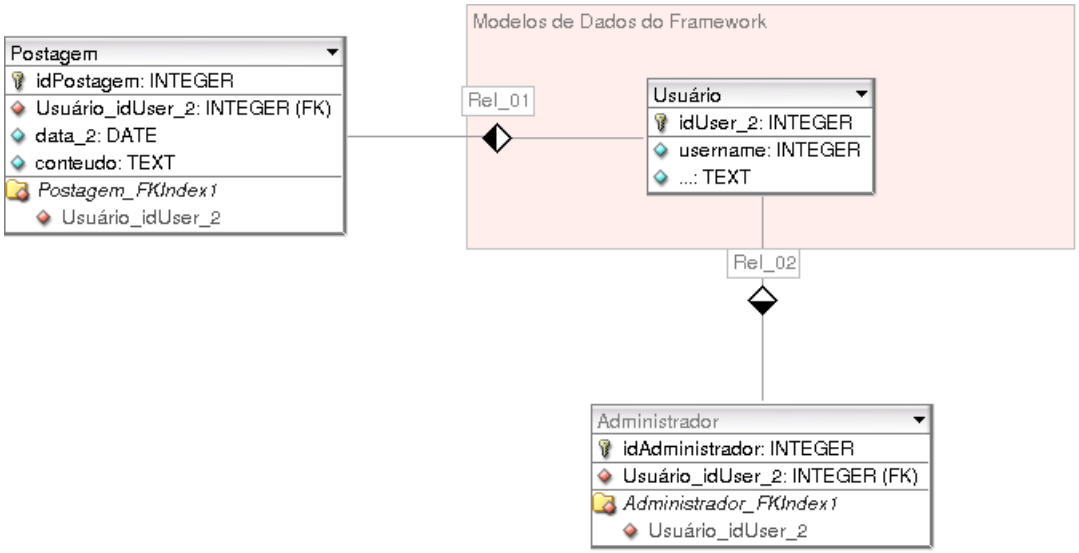
ANEXO A – Diagrama Requisição/Resposta Django



APÊNDICE A- Caso de uso do sistema teste



APENDICE B - Modelagem de dados



APENDICE C – Prototipação do sistema teste