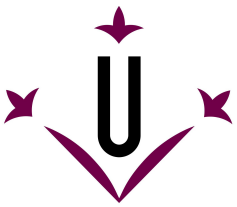

Universitat de Lleida
Escola Politècnica Superior
Màster en Enginyeria de Programari Lliure

Treball de final de màster

Estudi de la plataforma Google App Engine utilitzant Django

Autor: Josep Maria Brunetti Fernández
Director: Juan Manuel Gimeno Illa
Juliol de 2009

Pròleg

La idea d'aquest treball neix a l'abril del 2008, quan es publica la primera versió de Google App Engine en Python. Durant l'estiu, vaig començar a fer petites proves amb la plataforma i també amb Django. A principis de 2009 és quan em plantejo el tema sèriament, començo a documentar-me i a treballar en l'aplicació.

A finals del Maig de 2009 vaig poder publicar la primera versió de l'aplicació a Google App Engine. Des de llavors, he estat corregint errades, afegint noves funcionalitats i optimitzant l'aplicació.

Es fa dificil donar les gràcies a tota la gent que ha fet possible aquest treball. Per molt que expliqués, mai sabria com fer-ho. Tot i això, vull donar gràcies en primer lloc al Juan Manuel Gimeno, director d'aquest treball. Des d'un primer moment es va interessar pel tema i m'ha guiat tant amb el desenvolupament de l'aplicació com amb la realització de la memòria. Sense ell, res d'això seria possible.

També he d'agrair especialment l'interés i l'ajuda del Pol amb l'aplicació. No només per les proves i suggerències, sinó també per la galeria de fotos i alguna altra aportació de codi.

Gràcies també a tota la gent que ha provat l'aplicació, m'ha donat la seva opinió i m'ha permés corregir errors: Arnau (t'he fet servir d'exemple), Josep, Judit, Albert... i tots els noms que em deixo.

Si aquest document està mínimament ben escrit és també gràcies a l'Isabel i les seves correccions. Espero que no t'hagis avorrit gaire.

A tota la meva família, gràcies per donar-me suport i animar-me durant aquest treball i també tots aquests anys d'universitat. Sé que moltes vegades no enteneu perquè dedico tan temps als ordinadors però sempre ho heu respectat.

I a tots aquells que d'una manera o altra m'han ajudat, tot i que el seu nom potser no aparegui explícitament en aquestes línies, moltíssimes gràcies. Espero i desitjo que aquesta aplicació us sigui útil algun dia.

Índex

Índex de figures

Índex de taules

Capítol 1

Introducció

1.1 Objectius

L'objectiu principal d'aquest treball és estudiar la plataforma Google App Engine, en la seva versió en Python. Des que es va començar aquest treball, s'han publicat moltes millores i nous serveis de la plataforma i alguns temes no s'han pogut estudiar per falta de temps. En aquest treball s'estudia el nucli de la plataforma, la seva base de dades i les APIs principals.

Google App Engine proporciona un framework per a desenvolupar aplicacions web, però aquest és molt limitat. Aquesta mancança es soluciona amb Django, un framework per a desenvolupar aplicacions web de forma ràpida, que es pot executar sobre Google App Engine. Per això, estudiar Django és el segon objectiu d'aquest treball.

La millor manera d'estudiar el funcionament de Google App Engine i Django és desenvolupant una aplicació real que intenti aprofitar les característiques de la plataforma. Per aquest motiu, el tercer objectiu del treball és desenvolupar una aplicació web multiusuari, de codi lliure, que permeti crear blogs orientats a la publicació i seguiment de viatges.

Les característiques de l'aplicació fan que a priori sigui una aplicació que permetrà provar el rendiment i escalabilitat de Google App Engine. Aquesta aplicació es basa sobretot en la publicació de fotografies i mapes dels viatges, i per això haurà d'integrar serveis gratuïts de Google com Picasa Web Albums, Google Maps o Google Earth.

A banda d'aquests tres objectius principals, n'hi ha d'altres de secundaris:

- Estudiar i utilitzar Google Code, una eina de gestió de versions, wiki, *issues*...
- Estudiar i integrar altres tecnologies de Google: Google Maps, Picasa Web Albums, Google Data APIs, etc.
- Utilitzar i integrar altres eines de programari lliure ja existents.

1.2 Motivacions

A l'abril de 2008 es va publicar la primera versió beta de Google App Engine. Recordo que aquell mateix dia vam comentar la notícia amb alguns companys de feina. Per la tarda, a classe d'Enginyeria del Software també va sortir el tema i ho vam comentar amb el Juan Manuel. En arribar a casa vaig començar a llegir la documentació i vaig fer els primers tutorials. Des d'aquell moment vaig tenir clar que com a treball final del màster volia fer quelcom relacionat amb Google App Engine.

Pel que fa a l'aplicació que es pretén desenvolupar sobre Google App Engine, la principal motivació és personal. En els últims anys he tingut la sort de poder viatjar, i abans de marxar de vacances, amics i companys de feina sempre em pregunten si tinc algun blog on publicar el que faig durant tots aquests dies. Fins ara la meva resposta era "no".

Sovint, la solució per a molta gent és un blog convencional amb algun plugin per mostrar mapes i publicar fotografies en àlbums de Flickr o Picasa. Una altra opció és fer servir algun dels serveis ja existents que permeten crear blogs de viatges. En el meu cas, cap d'aquestes opcions em convenç ni satisfà les meves necessitats.

És en aquest moment quan se m'acudeix fer la meva pròpia aplicació per a publicar viatges i integrar-ho amb els complements que m'interessen. Així és com neixen molts projectes de programari lliure, amb el que s'anomena "Scratch an Itch" (gratar-se una picada), quan algú desenvolupa un programa en concret que li soluciona alguna necessitat.

Una altra motivació personal és poder iniciar un projecte de programari lliure pràcticament des de zero i aplicar els coneixements que he après en aquests dos anys de màster. A més a més, el fet de poder desenvolupar amb Python també és una motivació extra ja que és un llenguatge que m'agrada i que fins ara no he pogut utilitzar gaire.

Finalment, el fet de veure que amics i companys s'interessaven pel projecte, em donaven noves idees i provaven l'aplicació un cop publicada; em motiva a continuar-la desenvolupant un cop acabat aquest treball.

1.3 Estructura del document

Aquest document està dividit en set capítols, seguits d'un anex que amplia la informació i finalment la bibliografia.

El primer capítol és introductori i s'exposen els objectius i les motivacions d'aquest treball.

En el segon capítol es tracta l'estat de l'art, explicant el fenòmen del cloud computing i alternatives tecnològiques a Google App Engine. També es fa un repàs als blogs de viatges i altres projectes semblants.

El tercer capítol correspon a les tecnologies i llenguatges emprats en la realització d'aquest treball. En aquest treball hi ha moltes tecnologies implicades i en aquest capítol només es fa una petita introducció a elles. Les dues tecnologies principals, Django i Google App Engine, tenen capítols dedicats.

En el quart capítol es presenta Django, un framework escrit en Python per al desenvolupament ràpid d'aplicacions web. La documentació de Django és molt extensa i en aquest capítol es presenten les funcionalitats principals.

El cinquè capítol és el més important de tots. En ell es profunditza en la plataforma Google App Engine: les seves principals característiques, la base de dades que ofereix, les seves APIs i serveis, etc. En aquest treball només s'ha estudiat la versió Python de Google App Engine.

El sisè capítol és sobre la gestió del projecte. En aquest capítol s'explica el sistema de gestió de versions que s'ha utilitzat, com s'ha fet la documentació del projecte i aspectes legals del programari utilitzat.

El setè capítol correspon al cas pràctic d'aques treball. En ell s'exposen temes d'enginyeria del software i desenvolupament d'aplicacions web relacionats amb l'aplicació.

En l'annex adjunt es compara la tecnologia utilitzada en aquest treball amb una possible equivalència en Java. La informació d'aques annex s'ha tret d'un treball fet per a l'assignatura optativa Plataformes de Comerç Electrònic.

Finalment, s'inclou la bibliografia consultada durant aquest treball i la llicència d'aquest document.

NOTA: La majoria dels exemples de codi que s'inclouen en aquest treball s'han tret de l'aplicació desenvolupada. En altres casos, els exemples provenen de la documentació de Django i de la documentació de Google App Engine. En tots ells, el codi s'ha reduït per tal de fer-lo més simple i entenedor.

Capítol 2

Estat de l'art

2.1 Cloud computing

El cloud computing (computació en el núvol) és la tendència a oferir serveis de computació a través d'Internet. Els avantatges del cloud computing són evidents: permet a les aplicacions escalar ràpidament, en funció de les necessitats, sense necessitat d'afegir més maquinari ni conèixer la infraestructura o els detalls del funcionament. En contrapartida, el principal desavantatges és la dependència total del proveïdor.

Són vàries empreses les que últimament ofereixen productes basats en el cloud computing: Google Apps i Google App Engine, Azure de Microsoft, Amazon Web Services, Force.com (Salesforce) o IBM Blue Cloud.

El cloud computing es pot dividir en tres nivells en funció dels serveis que s'ofereixen: infraestructura com a servei, plataforma com a servei i software com a servei.

2.1.1 Infraestructura com a servei (IaaS, *Infrasstructure as a Service*)

La idea bàsica es la d'externalitzar els servidors en comptes de tenir-los en un datacenter d'una empresa. Amb una infraestructura com a servei es té una solució que es basa en la virtualització i en la que es paga per consum de recursos: espai de disc utlitzat, temps de CPU, espai de base de dades, transferència de dades... L'avantatge principal d'aquesta solució és que s'obté una escalabilitat semi-automàtica, aconseguint més infraestructura i recursos segons es van necessitant.

En certa manera, les solucions d'intraestructura com a servei es poden equiparar amb els hostings tradicional però amb la capacitat d'escalar automàticament. Respecte a la plataforma com a servei, la infraestructura com a servei és una solució molt més flexible però també requereix més dels usuaris pel que fa a instal·lació, configuració i manteniment del software que s'utilitzi.

Un exemple d'infraestructura com a servei és Amazon Web Services.

2.1.2 Plataforma com a servei (PaaS, *Platform as a Service*)

La idea principal d'una plataforma com a servei és la quantitat de capes que ofereix a l'hora de desenvolupar aplicacions. No només es resol el problema de la infraestructura

hardware (maquinari, ample de banda, escalabilitat, disponibilitat...) sinó també vàries capes de la infraestructura software: sistemes operatius, servidors web i d'aplicacions, bases de dades...

Una plataforma com a servei resol més problemes que una solució que només ofereixi intraestructura, però també afegeix moltes més limitacions en el desenvolupament.

Aquests són els principals avantatges d'una plataforma com a servei respecte un hosting tradicional:

- Escalabilitat: no cal preocupar-se del hardware ni de retocar l'aplicació per a que sigui escalable. De tot això se n'encarrega la plataforma.
- Cost segons el consum: el preu és segons els recursos que s'utilitzin: si en un més l'aplicació té poques visites, el cost serà menor. En el cas de Google App Engine, cal pagar a partir d'unes determinades quotes.
- Integració amb la resta de la plataforma: l'aplicació pot integrar altres solucions de la plataforma. Per exemple, en Google App Engine es poden utilitzar comptes de Gmail per a l'autenticació d'usuaris.
- Administració remota: les aplicacions es poden administrar i monitoritzar remotament via web.
- Facilitat en el desplegament: posar l'aplicació en producció és tan fàcil com fer servir un script o programa que se n'encarrega.
- Alta disponibiliat: molts pocs hostings poden oferir una disponibilitat tan alta com Google o Amazon.

D'altra banda, també existeixen desavantatges o limitacions importants:

- Eines disponibles limitades: eines, llenguatges de programació, bases de dades, llibreries, etc. Les nostres aplicacions s'executen en l'entorn d'un altre i per tan estan restringides, en part, per a que siguin escalables.
- Dependència del proveïdor: migrar d'un proveïdor PaaS a un altre o a un hosting tradicional pot ser pràcticament impossible, principalment, perquè l'aplicació utilitza eines només disponibles dins la plataforma.

Google App Engine i Force són dos exemples de plataformes com a servei.

2.1.3 Software com a servei (SaaS, *Software as a Service*)

El software com a servei és el més conegut dels tres nivells del cloud computing. Té com a objectiu el client final que utilitza el programari per ajudar i millorar els processos de la seva empresa. El software com a servei habitualment s'ofereix per Internet, a través del navegador. El programari és propietat del proveïdor i pot ser molt divers: controlar un procés del negoci, la gestió de clients, gestió de projectes, un ERP...

Els usuaris només han d'utilitzar el sevei i no s'han de preocupar d'instal·lar, configurar, mantenir o actualitzar el programari. Un dels aspectes més importants, la seguretat i cumpliment de lleis relatives a la protecció de dades, també està a càrrec del proveïdor.

L'exemple més clar i conegut de software com a servei és Google Apps, un conjunt d'aplicacions i serveis de Google orientats a empreses a un preu reduït. Entre d'altres, Google Apps integra Gmail, Google Talk, Google Calendar i Google Docs.

2.2 Alternatives tecnològiques

2.2.1 Force.com

Tot i que és una de les empreses que es dediquen al cloud computing menys conegudes, Salesforce és la pionera en aquest sector. Primer va començar oferint el seu CRM (Customer Relationship Management) via web , després un mercat d'aplicacions per a complementar-lo. I l'últim servei que ofereix és Force.com, una plataforma com a servei.

Force.com permet desenvolupar aplicacions web que s'executen nativament a Force.com i permet integrar altres productes serveis de Salesforce. A més a més, Salesforce i Facebook van signar un acord de col·laboració per a que les aplicacions de Force.com també es puguin integrar amb Facebook.

Force.com ofereix tres versions diferents:

- Edició gratüita (*Free edition*) permet desenvolupar una aplicació de prova per a 100 usuaris amb moltes limitacions.
- Edició per a empreses (*Enterprise edition*) permet desenvolupar fins a 10 aplicacions i accedir al CRM de Salesforce. Té un preu de 50$ mensuals.
- Edició il·limitada (*Unlimited edition)* a més a més de la resta, permet un nombre il·limitat d'aplicacions, dóna suport 24x7, més espai de disc i de base de dades, etc. Té un preu mensual de 75$.

2.2.2 Amazon Web Services

Amazon Web Services són un conjunt de serveis de computació que ofereix Amazon a través d'Internet com a serveis web. Al llarg dels anys, Amazon ha adquirit un gran coneixement sobre el desenvolupament i manteniment d'aplicacions web escalables i, des del 2004, ofereix aquest coneixement als desenvolupadors.

Avui en dia, Amazon ofereix una col·lecció d'eines que permeten desenvolupar aplicacions web escalables. Aquestes són les principals:

- Amazon Simple Storage Service (Amazon S3): és un servei d'emmagatzematge escalable, que pot ser utilitzat per altres aplicacions o per usuaris finals. Permet allotjar fitxers de fins a 5GB i els preus a Europa són de 0,18 dòlars al mes per GB d'espai i 0.17 dòlars al mes per GB de transferència.

- Amazon Elastic Compute Cloud (Amazon EC2): és un servei de lloguer de màquines virtuals que permet el desenvolupament d'aplicacions web escalables. Entre d'altres, permet sistemes operatius com Linux, OpenSolaris i Windows Server 2003. Amazon EC2 permet al client crear noves instàncies de servidors en paral·lel en qüestió de minuts. El preu del servei depen de la configuració del servidor i del temps de CPU que es consumeix.

- Amazon Simple DB (Amazon SDB): és un servei web per a executar consultes sobre una estructura de dades, de forma fàcil i sense la complexitat de les bases de dades relacionals. Actualment, les primeres 25 hores de computació de cada mes són gratuïtes i a partir d'aquest temps es paga per consum de CPU i per dades transferides.

Individulament, cadascun dels serveis anteriors són infraestructura com a servei, però l'ús d'aquests serveis de forma conjunta pot arribar a proporcionar una plataforma.

2.2.3 Windows Azure

Windows Azure és el nom de la plataforma de serveis web de cloud computing de Microsoft. Aquesta nova aposta de Microsoft va en la línea de Amazon Web Services, no és un producte per a usuaris finals sinó per a desenvolupadors i empreses que vulguin desenvolupar els seus serveis sobre una plataforma de Microsoft.

Windows Azure és un servei encara molt nou (es va anunciar a l'octubre de 2008) i encara està en fase de proves. Actualment desenvolupar utilitzant altres tecnologies de Microsoft com .NET, Live, SQL, SharePoint i Dynamics CRM, però Microsoft assegura que suportaran tot tipus de llenguatges de programació i entorns de desenvolupament.

2.2.4 Comparació amb Google App Engine

Tot i que l'objectiu principal d'aquest treball és estudiar Google App Engine, és necessari fer una comparació mínima d'aquesta plataforma amb altres de semblants. És molt difícil comparar aquestes tecnologies sense haver-les provat i més quan permeten usos diferents. Tota comparació possible és a partir d'opinions, de la seva documentació i dels serveis que ofereixen.

Force.com és una plataforma orientada a empreses que vulguin desenvolupar les seves aplicacions i integrar-les amb la resta de serveis de SalesForce, sobretot el seu CRM. Segurament l'escalabilitat que ofereixen no arriba a la de Google App Engine, però és un servei molt més madur i suficient per a moltes empreses.

Amazon Web Services permet escollir entre varis serveis i integrar-los, mentre que Google App Engine és un paquet indivisible. En Google App Engine, l'escalabilitat és automàtica. En canvi, Amazon Web Services ofereix màquines virtuals sota demanda i l'escalabilitat depen en bona part de la programació de l'aplicació. Un punt a favor de Amazon és el seu sistema d'emmagatzematge de fitxers, S3. A dia d'avui, Google App Engine no ofereix cap servei d'aquest tipus.

Windows Azure és un servei massa nou i en fase de proves per a poder comparar-lo amb Google App Engine. Tan App Engine com Windows Azure aprofiten per lligar les aplicacions a la resta dels seus productes. Això és un avantatge respecte la resta, sobretot pel que fa a l'autenticació d'usuaris que ofereixen. Però si s'escolleix Windows Azure, el lligam amb Microsoft és total: és necessari tenir Windows, Visual Studio i altres programes propietaris per a poder fer servir el seu SDK. En canvi, amb Google App Engine es pot desenvolupar amb qualsevol sistema operatiu.

Un punt en el que Google App Engine guanya a la resta és en els preus. No hi ha cap barrera d'entrada: provar-ho és gratuït i ho segueix sent per usos bàsics. I quan l'aplicació supera les quotes, els preus que s'ofereixen continuen sent millors que els de la resta.

Cal tenir en compte que, amb qualsevol d'aquestes solucions, estem dipositan les nostres aplicacions en les mans de tercers i la dependència del proveïdor En alguns casos, migrar d'una plataforma a una altra o a un allotjament particular pot ser pràcticament impossible.

2.3 Blogs de viatges

En els últims anys, el fenòmen del blogging s'ha convertit en dels més populars a Internet. Existeixen moltes eines que permeten la creació i manteniment de blogs. Algunes ofereixen una solució completa i no requereixen cap allotjament o instal·lació (com Blogger) i d'altres consisteixen en una solució software que necessita d'una instal·lació (com WordPress o Movable Type).

Hi ha varis tipus de blog depenent del seu contingut i de la forma en que aquest es presenta: PhotoBlogs, MicroBlogs, AudioBlogs, VideoBlogs... Tot i el nombre d'eines que permeten crear diferents tipus de Blog, no n'existeix cap de programari lliure que ofereixi una solució completa per al seguiment de viatjes, que permeti la publicació per etapes o dies, geoposicionament i l'enllaç amb àlbums de fotografies.

2.4 Altres projectes semblants

TravelPod (http://www.travelpod.com)

TravelPod és segurament el servei més utilitzat per a publicar blogs de viatges. És un servei gratuït i permet personalitzar bastant els mapes marcant-hi rutes. El disseny dels blogs és força antic i no es pot personalitzar.

TravelBlog (http://www.travelblog.org)

És també un projecte amb bastants usuaris registrats. La publicació d'entrades es força completa ja que permet introduir fotografies enmig del text. La creació de mapes no és automàtica.

I am on the move (http://www.iamonthemove.com)

És un servei senzill amb funcionalitats bàsiques però molt fàcil d'utilitzar. Permet escollir entre vàries plantilles pel blog i cada usuari té una URL pròpia: http://usuari.iamonthemove.com.

Get JEALOUS (http://getjealous.com)

És el servei més complet i que permet més opcions: es pot personalitzar el disseny del blog, es poden crear mapes de forma interactiva, pujar fotografies i vídeos, etc. Hi ha una versió gratuïta amb limitacions i una versió de pagament que permet més opcions.

Comparació amb aquest projecte

Ara per ara, totes les aplicacions anteriors tenen més funcionalitats que aquest projecte. No obstant, aquest projecte té característiques que el diferencien de la resta:

- És un projecte de programari lliure: qualsevol persona pot col·laborar amb el projecte o muntar la seva pròpia aplicació.
- És totalment gratuït.
- L'aplicació és multiidioma: ara mateix està en català, castellà i anglès, però es pot traduir a altres idiomes amb molta facilitat.
- L'autenticació és amb comptes de Google.
- Permet importar fotografies des de Picasa Web Albums (més endavant s'oferirà el mateix amb Flickr).
- Les URLs són netes: tots els blogs, viatges i entrades tenen unes URLs netes i elegants que són fàcils de recordar.

Capítol 3

Tecnologia utilitzada

En aquest treball hi ha moltes tecnologies implicades. A les dues principals, Google App Engine i Django, se'ls ha dedicat un capítol en concret. La resta de tecnologies utilitzades s'expliquen breument en aquest capítol.

3.1 Python

Python és un llenguatge de programació interpretat (no es necessita compilar) creat per Guido van Rossum al 1991. Avui en dia, Python es desenvolupa com un projecte de codi lliure administrat per la Python Software Foundation. El nom del llenguatge prové de l'afició del seu creador pels humoristes britànics Monty Python.

Python és un llenguatge multiparadigma. En comptes de forçar als programadors a seguir un estil concret de programació, permet varis estils: programació orientada a objectes, programació estructurada i programació funcional. Python es caracteritza per la seva indentació, un tipatge dinàmic i gestió automàtica de la memòria.

Una de les característiques principals de Python és la seva sintaxis, que ràpidament salta a la vista. En Python els blocs de codi es delimiten mitjançant l'ús d'indentació i no pas amb claus. Això implica que els espais en blanc tenen significat i obliga a una correcta indentació, millorant així la llegibilitat del codi.

Python ve acompanat d'un interpret interactiu que permet agilitzar el desenvolupament de programes, provant idees i veient ràpidament el seu resultat sense necessitat d'escriure un fitxer de codi sencer. L'interpret de Python és una de les millors eines per aprendre el llenguatge.

A l'igual que Java, Python té una àmplia llibreria estàndard que l'acompanya. La llibreria inclou mòduls per a manipular expressions regulars, crear interfícies gràfiques, conectar-se a una base de dades i un llarg etcètera.

3.2 XHTML

XHTML (*eXtensible Hypretext Markup Language*) és un llenguatge de marcat que substitueix a HTML com a estàndard per a les pàgines web. La versió 1.0 de XHTML és

únicament una versió XML del HTML tradicional, amb les mateixes funcionalitats, però cumpleix les especificacions de XML.

Les diferències entre XHTML i HTML són menors. El requisit més important és que el document estigui ben format i que totes les etiquetes estiguin ben tancades, com requereix XML.

3.3 CSS

CSS (*Cascading Style Sheets* o fulls d'estil en cascada) és un llenguatge formal que s'utilitza per definir la presentació d'un document estructurat escrit en HTML o XML. El W3C (*World Wide Web Consortium*) és l'organisme encarregat de l'especificació dels fulls d'estil que serveix com a estàndard.

La idea que es troba darrera dels CSS és separar l'estructura d'un document de la seva presentació. Quan s'utilitza CSS, les etiquetes dels documents HTML o XML no proporcionen informació sobre com ha de ser la presentació, sinó que únicament marquen l'estructura del document. El corresponent full d'estil s'encarrega d'especificar com s'ha de mostrar aquella etiqueta: color, font, alineació del text, mida...

Alguns dels avantatges d'utilitzar CSS són:

- Es té un control centralitzat de la presentació d'una web que agilitza qualsevol tipus d'actualització.
- Els navegadors permeten als usuaris especificar el seu propi full d'estil i així s'augmenta l'accessibilitat. Per exemple: persones amb deficiències visuals poden configurar el seu propi full d'estil augmentant la mida del text o remarcant els enllaços.
- Una lloc web pot disposar de diferents fulls d'estil per diferents famílies de dispositius o fins i tot per a que l'usuari pugui escollir.

3.4 JavaScript

JavaScript és un llenguatge de programació interpretat que s'utilitza principalment en pàgines web. JavaScript permet l'orientació a objectes però aquesta no es basa en classes sinó en el paradigma de la programació orientada a prototips: les noves classes es generen estenen els prototips en els que es basen els objectes.

Tots els navegadors actuals interpreten el codi JavaScript integrat en les pàgines web. El codi JavaScript es pot incloure en un document HTML o en qualsevol codi que s'acabi traduint a HTML en el navegador del client. El codi JavaScript és visible i pot ser llegit per l'usuari Ja que s'executa en el navegador del client.

3.4.1 Prototype

Prototype és un framework escrit en JavaScript orientat al desenvolupament senzill d'aplicacions web. Tot i que Prototype té moltes aplicacions, habitualment s'utilitza per a treballar amb AJAX ja que simplifica molt el maneig de l'objecte `XMLHttpRequest`.

En aquest treball, Prototype s'ha fet servir precisament per a treballar amb AJAX.

3.4.2 jQuery

jQuery és un framework escrit en JavaScript orientat també al desenvolupament ràpid i senzill d'aplicacions web. jQuery té moltes similituds amb Prototype i tots dos frameworks tenen funcionalitats comunes.

En aquest treball, jQuery s'utilitza per als selectors d'elements DOM i per a la galeria de fotos.

3.5 AJAX

AJAX, acrònim de *Asynchronous JavaScript And XML* (JavaScript Asíncron i XML), és una tècnica de desenvolupament web per a crear aplicacions interactives o RIA (*Rich Internet Applications*). Aquestes aplicacions s'executen en el navegedor del client mentre es manté la comunicació amb el servidor en segon pla. D'aquesta manera és possible realitzar canvis a les pàgines sense necessitat de recarregar-les. AJAX permet augmentar la interactivitat, velocitat i usabilitat de les aplicacions.

AJAX és una tecnologia asíncrona: les dades es demanen al servidor i es carreguen en segon pla sense interferir en el comportament de la pàgina. Habitualment, javascript és el llenguatge en que s'executen les crides AJAX, l'accés a les dades és realitza mitjançant un objecte `XMLHttpRequest` i les dades es transmeten amb XML o JSON.

Frameworks com jQuery o Prototype faciliten l'ús d'AJAX i el maneig de l'objecte `XMLHttpRequest`.

3.6 YAML

YAML és un format de serialització de dades llegible per humans, inspirat en llenguatges com XML o Python. En un principi, YAML significava *Yet Another Markup Language* (Un altre llenguatge de marcat), però per tal de distingir el seu propòsic centrat en les dades i no en el marcat), es va canviar el significat per *YAML Ain't Another Markup Language* (YAML no és un altre llenguatge de marcat).

La sintaxis de YAML és força senzilla i està dissenyada de manera que sigui fàcilment llegible i que a la vegada sigui fàcilment mapejable als tipus de dades més comuns: llistes, hash i valors simples. L'estructura indentada de YAML i la seva aparença el fan un llenguatge molt apropiat per a definir fitxers de configuració.

Google App Engine utilitza yaml en dos fitxers:

- **app.yaml:** defineix la configuració de l'aplicació: nom, versió, runtime, mapeig de URLs, etc.
- **index.yaml:** defineix els índex de les entitats de la datastore.

3.7 Google Maps

Google Maps és un servei gratuït de cartografia de Google. Ofereix mapes i imatges reals provinents de satèl·lits, que es poden consultar des del web de Google Maps o incrustar-les a un altre lloc web a través de la seva API.

L'API de Google Maps proporciona als desenvolupadors diverses maneres d'integrar Google Maps en el seu lloc web. Existeixen tres variants de l'API: l'API JavaScript de Google Maps, l'API de Google Maps per a Flash i l'API de Google Static Maps. En aquest treball s'utilitza l'API en JavaScript, en la seva segona versió.

3.7.1 Ús de l'API

El següent exemple mostra com incloure un mapa en una pàgina HTML:

```
<html>
  <head>
      <title>Google Maps Example</title>
    <script type="text/javascript"
      src="http://maps.google.com/maps?file=api&amp;v=2&amp;key=abcdefg
        &sensor=false">
    </script>
  </head>

  <body onload="initialize()">
    <div id="map" style="width: 500px; height: 300px"></div>
      <script type="text/javascript">
      function initialize() {
        if (GBrowserIsCompatible()) {
          var map = new GMap2(document.getElementById("map"));
          map.setCenter(new GLatLng(37.4419, -122.1419), 13);
        }
      }
    </script>
  </body>

</html>
```

En aquest exemple, l'API de Google Maps es carrega amb aquest codi:

```
<script type="text/javascript"
  src="http://maps.google.com/maps?file=api&amp;v=2&amp;key=abcdefg
    &sensor=false">
</script>
```

On s'especifiquen alguns paràmetres:

- `v`: la versió de l'API. En aquest cas, la 2.
- `key`: la clau per a utilitzar l'API. S'ha de demanar una clau per a cada domini.
- `sensor`: aquest paràmetre permet que l'API intenti determinar l'ubicació de l'usuari.

Per a que el mapa es pugui mostrar en una pàgina web, necessita un lloc on ubicar-se. Habitualment això es fa utilitzant un element `div` amb un identificador (`id`) en concret. En aquest element també s'especifiquen les mides del mapa.

```
<div id="map" style="width: 500px; height: 300px"></div>
```

La classe que representa els mapes és `GMap2`. Al crear una instància d'aquesta classe, s'especifica l'element HTML de la pàgina que farà de contenidor:

```
map = new GMap2(document.getElementById("map"));
```

Un cop s'ha creat el mapa, cal inicialitzar-lo. La inicialització es realitza amb el mètode `setCenter()` que rep unes coordenades `GLatLng` i el nivell d'ampliació. És obligatori cridar aquest mètode abans de fer cap altre operació amb el mapa:

```
map.setCenter(new GLatLng(37.4419, -122.1419), 13);
```

Finalment, per a garantir que el mapa s'afegeixi quan la pàgina s'ha carregat per complet, s'executa la funció `initialize()` quan l'element `body` de la pàgina HTML ha rebut l'event `onload`:

```
<body onload="initialize()">
```

Els passos anteriors són els mínims per incloure un mapa en una pàgina web. Un cop el mapa està carregat i inicialitzat, es pot treballar amb ell i afegir-hi altres continguts.

Per exemple, el següent codi afegeix una marca al mapa que en fer clic desplega una finestra amb text informatiu:

```
var point = new GLatLng(37.4419, -122.1419);
var text = 'Aquest text pot contenir <b>html</b>'
var marker = new GMarker();
GEvent.addListener(marker, "click", function() {
  marker.openInfoWindowHtml(text);
});
map.addOverlay(marker);
```

El resultat d'aplicar tot aquest codi és el següent:

Aquest és només un petit exemple de l'ús de l'API de Google Maps. L'API és molt extensa, potent i té moltes més funcionalitats.

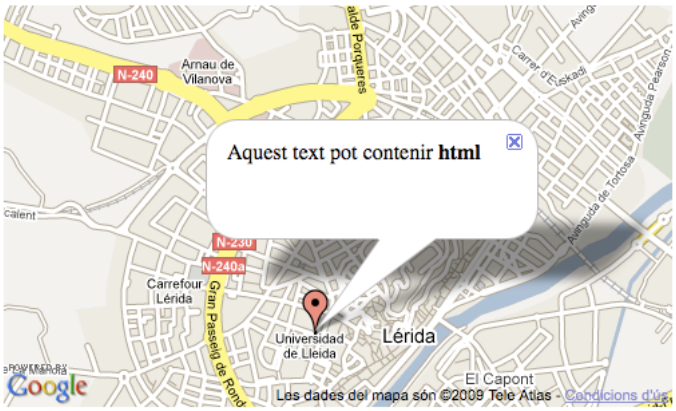

Figura 3.1: Exemple de Google Maps

3.8 Picasa Web Albums

Picasa Web Albums és un servei de Google que permet publicar fotografies i compartirles a través d'una interfície web. Aquest servei permet als usuaris amb un compte de Google compartir fotografies, fins a 1 GB d'espai. Els usuaris poden incrementar aquest espai des de 10 GB per 20$ anuals fins a 400GB per 500$ anuals.

Aquest servei permet pujar les fotografies a través d'una interfície web, de Picasa (un programa de Google per a organitzar i editar fotografies) o des de iPhoto de Mac OS. Altres programes de gestió de fotografies també tenen extensions per a publicar les fotos a Picasa Web Albums.

Picasa Web Albums té una API que permet treballar amb àlbums, fotografies, comentaris i altres elements.

Capítol 4

Django

4.1 Introducció

Django és un framework de desenvolupament web en python de codi obert. Originalment, els seus creadors feien servir el framework per administrar vàries pàgines web de notícies. Quan es va publicar sota llicència BSD al juliol de 2005, li van donar aquest nom pel guitarrista de jazz Django Reinhardt. Al juny de 2008 es va crear la Django Software Foundation per tal de promocionar, donar suport i seguir endavant amb el projecte.

4.2 Característiques

El principal objectiu de Django és facilitar la creació d'aplicacions web que emprin una base de dades. Django es centra en l'automatització i en reutilitzar els components, fent ènfasi en el principi DRY (*Don't Repeat Yourself*).

Les principals caracteristiques de Django són:

- **Mapeig Objecte-Relacional:** les classes del model es defineixen en python i es treballa amb elles amb l'API d'accés a base de dades. Això permet desenvolupar independentment del motor de base de dades utilitzat i evita l'ús de SQL.
- **Interfícies d'aministració automàtica:** proporciona una administració dinàmica i configurable que permet fer operacions CRUD (*Create, Read, Update, Delete*) sobre les classes del model. D'aquesta manera el programador pot centrar-se en la lògica de l'aplicació i oblidar-se dels continguts.
- **URLs elegants:** permet crear URLs elegants o netes fent servir expressions regulars. Això permet desenvolupar facilment serveis web REST (*Representational State Transfer*).
- **Sistema de plantilles:** utilitza un sistema de plantilles que permet separar disseny gràfic i programació. Es pot editar el HTML sense haver de tocar el codi Python.
- **Vistes genèriques:** incorpora un sistema de vistes genèriques per a fer tasques habituals: llistar registres, veure el detall d'un registre, esborrar un registre, etc.
- **Cache:** utilitza memcached per a obtenir un bon rendiment.

- **Aplicacions endollables (*pluggables*)**: les aplicacions es poden instal·lar en qualsevol altre projecte Django, són reutilitzables.
- **Internacionalització:** permet especificar cadenes de traducció per desenvolupar aplicacions multiidioma.

4.3 Projectes i aplicacions

Els projectes en Django es divideixen en aplicacions. Quan es crea una nova instal·lació de Django, a aquesta se l'anomena projecte. Un projecte de Django conté una o vàries aplicacions.

La diferència entre projecte i aplicació és la següent: una aplicació de Django és una aplicació web que fa una tasca concreta (un blog, un sistema d'enquestes, un catàleg...) i un projecte és una col·lecció d'aplicacions configurades d'una determinada manera en un únic lloc web. Un projecte pot contenir vàries aplicacions i una aplicació pot estar en varis projectes.

L'estructura d'un projecte en Django és la següent:

```
blogyourtravel/
    __init__.py
    manage.py
    settings.py
    urls.py
```

Aquest fitxers són:

- `__init__.py`: un fitxer buit que indica a l'interpret de Python que aquest directori s'ha de considerar com un paquet (*package*) de Python.
- `manage.py`: una utilitat de línea de comandes per interaccionar amb el projecte. En la propera secció es detalla el seu funcionament.
- `settings.py`: el fitxer de configuració per al projecte de Django. En una propera secció també s'explica detalladament.
- `urls.py`: el fitxer de configuració de URLs del projecte. També s'explica el seu funcionament més endavant.

Dins de cada projecte hi ha una o vàries aplicacions. Cada aplicació consisteix en un paquet de Python que segueix una estructura recomanada. Cada aplicació pot tenir el seu propi model, vistes i plantilles.

L'estructura habitual d'una aplicació és la següent:

```
blogyourtravel/
    travellog/
        __init__.py
        models.py
        views.py
```

Els noms dels fitxers indiquen quin és el seu propòsit:

- `__init__.py`: igual que en l'exemple anterior, un fitxer buit que indica que aquest directori és un paquet de Python.
- `models.py`: un fitxer on es defineix el Model de l'aplicació.
- `views.py`: un fitxer on es defineixen les vistes (corresponen al Controlador de l'arquitectura MVC) de l'aplicació.

Les plantilles d'una aplicació es poden incloure dins de la pròpia aplicació (per exemple en un directori `/templates`) o bé en un únic directori per a tot el projecte. Els directoris on es troben les plantilles és una de les opcions que es poden configurar en el fitxer `settings.py`.

El principal avantatge de dividir un projecte en aplicacions és que aquestes es poden reutilitzar en altres projectes. Django porta vàries aplicacions instal·lades per defecte, per exemple:

- `django.contrib.auth:` un sistema d'autenticació a partir d'usuaris, grups i permisos.
- `django.contrib.admin:` una interfície web d'administració automàtica que permet afegir, editar i eliminar objectes de la base de dades.
- `django.contrib.sessions:` una aplicació per gestionar sessions.

Les aplicacions de Django es diu que són endollables (*plugables*) ja que es poden copiar en qualsevol instal·lació de Django. Existeixen moltes aplicacions de Django disponibles al lloc web www.djangoplugables.com.

En el cas de l'aplicació desenvolupada en aquest treball s'ha creat una única aplicació de Django, `travellog`, que permet crear els blogs de viatges. Si es vulgués crear, per exemple, un apartat de notícies al web on mostrar les últimes novetats o funcionalitats, es podria crear una nova aplicació de Django i anomenar-la `news`.

4.4 Comandes de Django

La manera de treballar amb Django és mitjançant scripts que permeten gestionar els projectes. Aquests scripts permeten fer tasques com: crear nous projectes i aplicacions, validar-ne els models, interactuar amb la base de dades, executar el servidor de desenvolupament, etc.

En concret, Django té dos scripts per administrar els projectes:

- `django-admin.py` és una utilitat de Django per a línia de comandes que es troba dins del `path` del sistema un cop s'ha instal·lat Django.
- `manage.py` és un fitxer que es crea automàticament en cada projecte i que delega al fitxer anterior però configurant prèviament alguns aspectes:

- Afegeix el paquet del projecte al `path` de Python (`sys.path`).
- Defineix la variable d'entorn `DJANGO_SETTINGS_MODULE` per a que apunti al fitxer `settings.py` del projecte.

Habitualment, el script `django-admin.py` s'utilitza per a crear nous projectes i aplicacions mentre que el script `manage.py` s'utilitza per la resta de tasques.

Algunes de les tasques que es poden fer amb aquests scripts són:

```
# Crear un nou projecte de Django
django-admin.py startproject blogyourtravel

# Crear una nova aplicació de Django
django-admin.py startapp travellog

# Validar els models de les aplicacions instal·lades
manage.py validate

# Crear les taules necessàries per a les aplicacions instal·lades
manage.py syncdb

# Llençar un interpret de Python per a provar les APIs de Django
manage.py shell

# Arrancar un servidor web lleuger per a fer proves
manage.py runserver

# Crear o actualitzar els fitxers de missatges per a un determinat locale
manage.py makemessages --locale=<locale>

# Compilar els fitxers de missatges
manage.py compilemessages

# Crear els models a partir de la base de dades
manage.py inspectdb

# Mostrar l'ajuda
manage.py --help
```

Moltes d'aquestes comandes permeten l'ús de paràmetres per a variar el seu funcionament. Django també permet definir noves comandes personalitzades.

4.5 Fitxer de configuració

Tota la configuració d'un projecte de Django es guarda en un fitxer que, per convenció, normalment s'anomena `settings.py` i es guarda a l'arrel del projecte. Quan s'utilitza Django s'ha de definir la variable d'entorn `DJANGO_SETTINGS_MODULE` que indica quin fitxer de configuració s'està fent servir.

En aquest fitxer de configuració es poden configurar moltes opcions de Django, per exemple:

```
DEBUG = True # Indica si es vol treballar en mode Debug
USE_I18N = True # Indica si es vol habilitar el suport I18N
```

```
LANGUAGE_CODE = 'en' # Indica l'idioma per defecte
LANGUAGES = (
    ('ca', 'Català'),
    ('es', 'Español'),
    ('en', 'English'),
) # Indica els idiomes disponibles

# Dades per a la connexió amb la base de dades
DATABASE_ENGINE = 'sqlite3'
DATABASE_NAME = 'dbname'
DATABASE_USER = 'user'
DATABASE_PASSWORD = 'password'
DATABASE_HOST = ''
DATABASE_PORT = ''
```

En aquest fitxer es poden definir també variables per a les aplicacions i, com que el fitxer de configuració és en Python, es poden utilitzar sentències de control. Per exemple:

```
if os.environ.get('SERVER_SOFTWARE','').startswith('Devel'):
    GMAPS_KEY = 'ABQIAAAAzk0rRtu0DY74pMSuZvzjBxTwM0brOpm-All5BF6PoaKBxRW...'
else:
    GMAPS_KEY = 'ABQIAAAAzk0rRtu0DY74pMSuZvzjBxQIxg0vkGdHr8GsIUrHs5Zwjs6...'
```

En l'exemple anterior, la variable `GMAPS_KEY` tindrà un valor diferent segons si l'aplicació s'està executant en l'entorn de desenvolupament en local o bé en l'entorn de producció. En aquest cas, `GMAPS_KEY` emmagatzema la clau per a fer servir la API de Google Maps per a un determinat domini: localhost o blogyourtravel.appspot.com.

A la documentació de Django es poden trobar totes les opcions disponibles [1].

4.6 Arquitectura

Django es basa en el patró d'arquitectura Model Vista Controlador (MVC), tot i que els seus desenvolupadors interpreten el patró de manera diferent i prefereixen anomenar-ho Model Template View (MTV). El controlador passa ha anomenar-se vista i la vista s'anomena plantilla.

En Django, una vista descriu les dades que es presenten a l'usuari però no necessariament el seu aspecte. Una vista habitualment delega les dades a una plantilla que descriu la forma de presentar-les. Es diu que el controlador de un patró MVC clàssic estaria representat pel propi Framework.

4.6.1 Model

Un model en Django és la descripció d'un tipus de dades de la nostra aplicació, que es guardarà en base de dades. Dit d'una altra manera, són les classes que necessitem persistir.

[1] http://docs.djangoproject.com/en/dev/ref/settings/

Cada model es representa amb una classe escrita en Python, que estén la classe `django.db.models.Model`. La classe base, `Model`, conté tots els mètodes necessaris per a fer que els objectes sigui capaços d'interactuar amb la base de dades i deixa que les classes del Model només s'hagin d'ocupar de definir els seus atributs. Habitualment, cada classe del Model correspon a una taula de la base de dades i cada atribut a una columna d'aquesta taula.

La part més important d'una classe del Model són els seus atributs. Cada atribut és una instància d'una de les subclasses de la classe `django.db.models.Field`. Django utilitza el tipus d'aquestes classes per a:

- Conèixer el tipus de columna de la base de dades(`Varchar, Text, Datetime`...)
- Crear la intefície d'administració automàtica.
- Validar el tipus de dades en formularis i en l'administració.

Existeixen tipus de camps per a representar els tipus de dades més comuns, per exemple:

Classe en Django	**Tipus en python**	**Tipus en MySQL**	**Representació en HTML**
BooleanField	bool	TINYINT	<input type="checkbox">
CharField	str	VARCHAR	<input type="text">
DateField	datetime.date	DATE	<input type="text">
DecimalField	float	DECIMAL	<input type="text">
IntegerField	int	INT	<input type="text">
TextField	str	TEXT	<textarea>

Taula 4.1: Tipus de dades en Django i equivalències

A la documentació de Django es pot trobar el llistat complet de tipus d'atributs[2]. A més dels tipus que proporciona Django, es poden definir nous tipus.

Cada tipus d'atribut té els seus propis arguments. Per exemple, el tipus `CharField` (i les seves subclasses) té un argument obligatori, `max_length`, que especifica la mida del camp `Varchar` de la base de dades. A part dels arguments propis de cada tipus, també n'hi ha de comuns per a tots ells.

```
from django.db import models

class Comment(models.Model):
    name = models.CharField(max_length=100)
    email = models.EmailField(required = False)
    comment = models.TextField()
    accepted = models.BooleanField(default = False)
```

Si el model no té un atribut que sigui clau primària, Django n'assigna una autonumèrica accessible mitjançant l'atribut `id`.

[2]http://docs.djangoproject.com/en/dev/ref/models/fields

4.6.1.1 Relacions

Les bases de dades relacionals permeten relacionar les taules entre elles. Django també permet modelar els tres tipus de relacions més comunes:

1. **Relacions many-to-one:**
   Per a definir relacions many-to-one s'utilitza `ForeignKey` de la mateixa manera que s'utilitzava qualsevol subclasse de `Field`. Al tipus `ForeignKey` se li passa com a argument la classe amb la que es vol relacionar:

```
from django.db import models

class Travel(models.Model):
    # ...

class Entry(models.Model):
    travel = models.ForeignKey(Travel)
    # ...
```

2. **Relacions many-to-many:**
   Per a definir relacions many-to-many també s'utilitza `ForeignKey` igual que en l'apartat anterior:

```
from django.db import models

class Tag(models.Model):
    # ...

class Travel(models.Model):
    tags = models.ManyToManyField(Travel)
    # ...
```

   És indiferent en quina de les dos classes es defineix la relació many-to-many, el resultat és el mateix, però només cal definir-ho en una de les dos. En aquest cas, és més natural pensar en un viatge que té varis tags que no pas en un tag que apareix en varis viatges.

3. **Relacions one-to-one:**
   Les relacions one-to-one també es defineixen com en els casos anteriors amb el tipus `OneToOneField`. Conceptualment són similars a una clau forana amb un índex `UNIQUE` (en django: `ForeingKey` amb `unique=True`).

4.6.1.2 Interacció amb el Model

Un cop creades les classes del Model, Django automàticament proporciona una API que permet interaccionar amb el Model sense necessitat d'escriure SQL. Django també permet crear sentències SQL tot i que amb la seva API es poden realitzar la major part d'operacions necessàries.

Crear objectes

Per a crear nous objectes només cal crear una nova instància d'una classe i cridar al mètode `save()` per a guardar-la. Els atributs es poden especificar en el mètode constructor o accedint als atributs de la classe.

```
from myproject.myapp.models import Comment
c = Comment(name='My name', email='My email')
c.comment = 'My comment'
c.save()
```

Modificar objectes

Si el mètode `save()` es crida sobre un instància ja existent en la base de dades, aquesta s'actualitza en comptes de crear-ne una de nova.

```
c.accepted = True
c.save()
```

Esborrar objectes

Per a esborrar un objecte del model s'utilitza el mètode `delete()`

```
c.delete()
```

Obtenir objectes

Per a obtenir tots els registres d'un model s'utilitza el mètode `all()`.

```
comments = Comment.objects.all()
for comment in comments:
    print comment.name
```

L'atribut `objects` és un atribut especial que s'utilitza per a fer consultes a la base de dades. Aquest atribut es diu que és un `manager`. Tots els models reben un `manager` amb nom `objects`, que s'utilitza per a cercar sobre el model.

En l'exemple anterior, el mètode `all()` retorna un `QuerySet` amb totes les files de la taula de la base de dades. Un `QuerySet` representa una col·lecció d'objectes de la base de dades que pot ser filtrada. Internament, un `QuerySet` és construeix sense interaccionar amb la base de dades i només es fa la consulta quan el `QuerySet` s'evalua. En l'exemple anterior, el `QuerySet` s'evalua en recórrer el bucle.

Filtres

Sobre el `manager` es poden aplicar filtres per a obtenir uns objectes en concret. Per a aplicar filtres s'utilitzen dos mètodes:

- `filter(**kwargs)`: retorna aquells objectes que coincideixen amb els paràmetres passats.
- `exclude(**kwargs)` retorna aquells objectes que no coincideixen amb els paràmetres passats.

Cada filtre rep arguments amb clau indicant l'atribut sobre el que es vol filtrar i el valor que ha de tenir. Per exemple:

```
comments = Comment.objects.filter(accepted=True)
```

Els filtres també es poden combinar:

```
comments = Comment.objects.filter(accepted=True).exclude(email='aa@bb.com')
```

Obtenir un únic objecte

Els mètodes anteriors serveixen per obtenir varis objectes que es poden tractar com una llista. A vegades interessa obtenir un únic objecte, i per fer-ho s'utilitza el mètode `get()`. Aquest mètode obté un únic resultat i si la consulta retorna més d'un objecte llença una excepció.

```
comment = Comments.objects.get(id=3)
```

Els mètodes anteriors són els més bàsics de la API de Django. A banda d'aquests, n'hi ha d'altres per a fer filtres més avanats (amb sentències `LIKE` de SQL), per actualitzar o esborrar un conjunt d'objectes a la vegada, per a ordernar els resultats, etc. Aquests no s'han tractat ja que en aquest treball s'utilitza el Model de Google App Engine i no pas el de Django.

4.6.2 View

4.6.2.1 Gestió de URLs

Avui en dia, un dels factors a tenir en compte a l'hora de desenvolupar una aplicació web és la forma que tindran les URLs. Unes URLs netes i "amistotes" són més fàcils d'indexar pels cercadors i a la vegada són més fàcils de recordar i entendre pels usuaris del web. Django permet dissenyar les URLs de la manera que es vulgui, sense cap limitació ni necessitat de que acabin amb '.php', '.html', etc.

Com a URL neta s'enten una direcció sense paràmetres GET, per exemple:

http://blogyourtravel.appspot.com/usuari1

Mentre que una URL "bruta" seria semblant a:

http://blogyourtravel.appspot.com/?user=usuari1

Per a dissenyar les URLs d'una aplicació Django es crea una espècie de taula que mapeja patrons de URLs (expressions regulars) a functions de Python que s'executaran (les vistes). Aquest mapeig es fa en el mòdul URLconf i permet que les URLs estiguin desacoplades de la resta de l'aplicació.

```
from django.conf.urls.defaults import *

urlpatterns = patterns('',
    (r'^register/$', 'travellog.views.register'),
    (r'^(?P<blog_key>[^\.^/]+)/$', 'travellog.views.blog_main'),
    (r'^(?P<blog_key>[^\.^/]+)/(?P<travel_key>[^\.^/]+)/$',
         'travellog.views.travel_detail'),
    (r'^(?P<blog_key>[^\.^/]+)/map/$', 'travellog.views.global_map')
)
```

Quan Django rep una petició, per exemple '/register/', fa el següent:

- Django determina quin és el mòdul URLconf que ha d'utilitzar, que s'especificia al fitxer de configuració settings.py.
- Django carrega aquest mòdul i busca la variable urlpatterns.
- Django esborra la barra del principi i es queda amb 'register/'. No es tenen en compte els paràmetres GET o POST.
- Django comprova seqüencialment els patrons de URLs i s'atura al primer que coincideix amb la URL.
- Quan l'expressió regular coincideix, Django importa i crida la vista que correspon, que és una simple funció en Python. A la vista se li passa un paràmetre `HttpRequest` i la resta de paràmetres que s'han capturat amb l'expressió regular.

En aquest cas, la petició coincidiria amb la primera expressió regular i es cridaria a la funció

`travellog.views.register`.

Com que aquesta URL no captura cap paràmetre, es cridaria a la funció passant-li només la petició (`request`). En canvi, la petició '/usuari1/viatge1' coincidiria amb la tercera expressió regular i es cridaria a la funció

```
travellog.views.travel_detail(request,
blog_key='usuari1', entry_key='viatge1')
```

El mòdul gestor de URLs permet assignar un prefix a totes les vistes, per exemple 'travellog.views' i també permet incloure configuracions d'altres fitxers.

4.6.2.2 Vistes

Les vistes de Django són funcions en Python que equivalen als controladors d'una arquitectura MVC tradicional. Una vista rep una petició (`request`) i retorna una resposta (`response`). La resposta pot ser el contingut HTML d'una pàgina, un `redirect`, un error 404, una imatge... La funció pot tenir la lògica necessària i el codi pot provindre de qualsevol lloc sempre i quan estigui en el Python path.

Un exemple d'una vista senzilla és el següent:

```
from django.http import HttpResponse
import datetime

def current_datetime(request):
    now = datetime.datetime.now()
    html = "<html><body>It is now %s.</body></html>" % now
    return HttpResponse(html)
```

Aquesta funció calcula la data actual i retorna un objecte de la classe `HttpResponse` que conté la resposta HTML. Totes les vistes han de retornar un objecte de la classe `HttpResponse` o d'alguna de les seves subclasses com `HttpResponseNotFound`, `HttpResponseServerError`, `HttpResponseRedirect`, etc.

4.6.3 Template

Django proporciona un sistema de plantilles que permet separar la programació del disseny. El sistema és semblant a les plantilles Smarty en PHP o Cheetah en Python.

El sistema de plantilles de Django es basa en el principi de l'herència: tot està definit en una plantilla base i la resta de plantilles extenen aquesta plantilla base. Això permet definir l'estructura o esquelet en una plantilla base que conté els elements comuns per a totes les plantilles. En la plantilla base també es defineixen blocs que la resta de plantilles poden sobreescriure.

La manera més senzilla d'entendre el funcionament d'aquests sistema és amb un exemple:

```
<html>
<head>
    <title>{% block title %} My Title {% endblock %}</title>
</head>

<body>
    <div id="sidebar">
        {% block sidebar %}
        <ul>
            <li><a href="/">Home</a></li>
            <li><a href="/blog/">Blog</a></li>
        </ul>
        {% endblock %}
    </div>

    <div id="content">
        {% block content %}{% endblock %}
    </div>
</body>
</html>
```

Aquesta plantilla, 'base.html', defineix l'esquelet HTML del nostre lloc web. L'estructura és de dos columnes, una amb el menú i l'altra amb els continguts. La resta de plantilles s'encarregaran de ompliar aquest bloc pels continguts.

4.6.3.1 Carregar plantilles a les vistes

En l'exemple de l'apartat anterior (Vistes), la vista retornava una resposta HTTP amb tot el codi HTML *hard-coded*. Aquesta no és la filosofia de Django ja que si es vol canviar la maquetació de la pàgina cal editar el codi Python. Per tal de separar el disseny gràfic del codi s'utilitza el sistema de plantilles de Django:

```
from django.template.loader import get_template
from django.template import Context
from django.http import HttpResponse
import datetime

def current_datetime(request):
    now = datetime.datetime.now()
    t = get_template('current_datetime.html')
    html = t.render(Context({'current_date': now}))
    return HttpResponse(html)
```

Aquest codi carrega la plantilla 'current_datetime.html' i se li passa un context. El context és un diccionari que mapeja variables de la plantilla a variables de Python. La plantilla 'current_datetime.html' seria la següent:

```
<html>
    <body>
        <p>It is now {{ current_date }}</p>
    </body>
</html>
```

Les etiquetes entre són variables de la plantilla que es substituiran per variables del context. En aquest cas, l'etiqueta `current_date` és substituirà per la variable Python `now`

4.6.3.2 Etiquetes i filtres

El sistema de plantilles de Django permet l'ús d'etiquetes (`tags`) i filtres (`filters`) que serveixen per mostrar codi dinàmic i aplicar modificadors a les variables respectivament.

Filtres (`filters`)

Els filtres serveixen per a modificar les variables a l'hora de mostrar-les. La seva sintaxis és {{ variable|filtre }} i es poden encadenar: {{ variable|filtre1|filtre2 }}

Aquests són alguns dels filtres que s'han utilitzat:

- `date`: dóna format a una data.

  ```
  {{ value|date:"D d M Y" }}
  ```

- `length`: retorna la mida d'una variable.

  ```
  {{ value|length }}
  ```

- `lower`: converteix una cadena de text a minúscules.

  ```
  {{ value|lower }}
  ```

Etiquetes (`tags`)

Les etiquetes de Django permeten mostrar codi dinàmic en un pseudo-Python. Les etiquetes tenen l'aspecte {% etiqueta %} o {% etiqueta %} continguts {% etiqueta %}.

Aquestes són algunes de les etiquetes que s'han utilitzat:

- `for`: per a iterar sobre un array:

```
{% for object in object_list %}
    <p>{{ object.title }}</p>
{% endfor %}
```

- `comment`: s'ignora el contingut entre les etiquetes:

```
{% comment %}
    Aquest bloc és un comentari i no es processarà
{% endcomment %}
```

- `if`: evalua si una variable existeix o bé està buida:

```
{% if object_list %}
    Number of objects: {{ object_list|length }}
{% else %}
    No objects.
{% endif %}
```

- `ifequal`: s'avalua el contingut del bloc si els dos arguments són iguals

```
{% ifequal user.username "josepb" %}
    ...
{% endifequal %}
```

També es poden crear filtres i etiquetes personalitzades. La llista completa d'etiquetes i filtres es pot consultar a la documentació de Django [3].

4.7 APIs i utilitats

4.7.1 Funcions d'accés directe (`shortcuts`)

En l'apartat en que es parla de plantilles de Django s'ha vist com es carregava una plantilla, se li passava un context i es retornava la resposta. El codi per a fer això és força llarg i per això Django proporciona algunes funcions d'accés directe (`shortcut`).

Algunes de les funcions d'accés directe que s'han utilitzat són:

- `render_to_response`: carrega una plantilla amb un context i retorna la resposta HTTP corresponent.

[3] http://docs.djangoproject.com/en/dev/ref/templates/builtins/

```
from django.shortcuts import render_to_response

def current_datetime(request):
    now = datetime.datetime.now()
    return render_to_response('current_datetime.html',
                              {'current_date' : now})
```

- `get_object_or_404`: carrega un objecte de la base de dades i si no existeix llença una excepció `django.http.Http404`. Aquesta excepció es capturada per Django i es mostra una plantilla '404.html' que es pot personalitzar.

```
from django.shortcuts import get_object_or_404

def blog_main(request, blog_key):
    blog = get_object_or_404(Blog, 'url =', blog_key)
    ....
```

4.7.2 Vistes genèriques

Sovint, moltes de les funcionalitats de les aplicacions web es basen en tasques monòtones i repetitives. Per exemple, en molts casos, el contingut que es presenta a l'usuari és un llistat d'ítems amb paginació i, quan s'accedeix a un ítem en concret, es mostra informació més detallada.

En aquestes situacions, les vistes que s'encarregarien de proporcionar aquesta funcionalitat serien pràcticament iguals. Per exemple, si comparéssim un llistat de notícies amb un llistat de productes, les úniques diferències entre les vistes serien: el nombre d'ítems a paginar, la llista d'objectes i la plantilla que els mostra.

Per a aquests casos, existeixen les vistes genèriques (`generic views`). Les vistes genèriques es van crear per agilitzar aquestes tasques repetitives i es basen en patrons comuns en el desenvolupament web.

A continuació es detallen les vistes genèriques més utilitzades en l'aplicació que s'ha desenvolupat.

Vista genèrica de llistat

La vista genèrica `django.views.generic.list_detail.object_list` representa una pàgina amb un llistat d'objectes. L'únic argument obligatori d'aquesta vista és `queryset` que ha de contenir un QuerySet amb els objectes a mostrar. Opcionalment, se li poden passar el nombre d'ítems a mostrar per pàgina, el nom de la plantilla a carregar, altres variables de context, etc. Si no s'especifica la plantilla, Django utilitza `<nom_del_model>_list.html`.

Aquest és un exemple d'una vista genèrica de llistat utilitzada:

```
def travel_list(request, blog_key):
    blog = get_object_or_404(Blog, 'url =', blog_key)
    travels = Travel.all().filter('blog=',blog).filter('published=',True)
                  .order('-start_date')
```

```
    return object_list(request, queryset = travels, paginate_by=10,
                       extra_context={'blog' : blog} )
```

En aquest exemple, les dues primeres línies dins de la funció són per accedir als models de Google App Engine. La funció de vista genèrica és l'última línea i se li passa un llistat d'objectes (`travels`), el nombre d'ítems per pàgina (10) i una variable extra de context (`blog`). La plantilla que carregarà la vista serà `travel_list.html` ja que el nom del model és `Travel`

Vista genèrica de detall

La vista genèrica `django.views.generic.list_detail.object_detail` representa una pàgina amb un únic objecte. Aquesta vista té dos arguments obligatoris:

- `queryset`: el QuerySet en el que es troba l'objecte.
- `object_id` o `slug`: si s'especifica el paràmetre object_id ha de ser el valor de la clau primària de l'objecte. Si s'utilitza el paràmetre `slug` també cal indicar el paràmetre slug_field.

Aquesta vista també pot tenir altres paràmetres opcionals com el nom de la plantilla a carregar, altres variables de context, el tipus MIME, etc. Si no s'especifica la plantilla, Django utilitza `<nom_del_model>_detail.html`.

Aquest és un exemple d'una vista genèrica de detall utilitzada:

```
def entry_detail(request, blog_key, travel_key, entry_key):
    blog = get_object_or_404(Blog, 'url =', blog_key)
    travel = get_object_or_404(Travel, 'slug =', travel_key)
    entry = get_object_or_404(TravelEntry, 'slug =', entry_key)

    return object_detail(request, TravelEntry.all(), slug=entry_key,
        slug_field='slug', extra_context={'blog': blog, 'travel': travel)
```

Django proporciona altres vistes genèriques que es poden consultar a la documentació [4].

4.7.3 Formularis

Els formularis HTML són bàsics en les aplicacions web per tal d'interactuar amb l'usuari. Tot i que és possible processar els formularis utilitzant la classe `HttpRequest`, emprar la llibreria de formularis de Django permet:

- Mostrar el formulari HTML generat automàticament.
- Comprovar si les dades enviades són correctes i si no ho són tornar a mostrar el formulari amb els errors.
- Convertir les dades enviades a tipus de Python.

[4] http://docs.djangoproject.com/en/dev/ref/generic-views/

Un formulari es representa amb una classe que hereta de la classe `django.forms.Form` i la forma de declarar els camps del formulari és molt semblant a la de les classes del Model.

A continuació es mostra un exemple d'un formulari de contacte d'un lloc web:

```
from django import forms

class ContactForm(forms.Form):
    name = forms.CharField(max_length=100, initial='Your name')
    message = forms.CharField(widget=forms.Textarea)
    sender = forms.EmailField(help\_text='Enter a valid email address')
    cc_myself = forms.BooleanField(required=False, label='Copy for myself?')
```

Un formulari es composa per un o més camps que corresponen a instàncies d'una de les subclasses de la classe `django.forms.Field`. En aquest cas, `CharField`, `EmailField` i `BooleanField` són els únics tipus de camps que utilitzem. Cada tipus de camp s'encarrega de validar les dades entrants, mostrar un widget adeqüat, convertir les dades a tipus de Python i validar-les.

A l'igual que en les classes del Model, els camps dels formularis es poden instanciar passant arguments que permeten configurar el widget en que es mostraran els camps, el seu label corresponent, el valor inicial, etc. En el cas de l'exemple anterior:

- El camp `name` podrà tenir un màxim de 100 caràcters i el seu valor inicial serà 'Your name'.
- El camp `message` es mostrarà amb un `<textarea>` en comptes de `<input type=''text''>`
- El camp `sender` mostrarà el text d'ajuda 'Enter a valid email address'.
- El camp `cc_myself` no serà obligatori i el seu label serà 'Copy for myself?' en comptes de 'cc_myself'.

4.7.3.1 Processar els formularis

La forma més habitual per a processar formularis en les vistes de Django és la següent:

```
def contact(request):
    if request.method == 'POST': # El formulari s'ha enviat
        # Es crea el formulari amb les dades enviades
        form = ContactForm(request.POST)
        if form.is_valid(): # El formulari és vàlid
            # Processar el formulari
            ...
            # Redirigir després del POST.
            return HttpResponseRedirect('/thanks/')
    else:
        form = ContactForm() # Es crea el formulari buit.

    return render_to_response('contact.html', {
        'form': form,
    })
```

El codi anterior té tres possibles camins:

1. Si el formulari no s'ha enviat (la petició no és un POST) es crea una instància de ContactForm buida i es passa a la plantilla. Els camps del formulari estaran buits o bé tindran un valor per defecte si s'ha especificat.

2. Si el formulari s'ha enviat es crea una instància de ContactForm amb les dades enviades. Si el formulari és vàlid, aquest es processa i es redirigeix a l'usuari a la pàgina corresponent.

3. Si el formulari s'ha enviat però és invàlid, es passa el formulari de nou a la plantilla. Aquest cop el formulari també contindrà les dades enviades i els errors.

Quan s'ha comprovat que el formulari és vàlid (`form.is_valid` retorna `True`), es pot processar sabent que les dades han passat una sèrie de regles de validació. En comptes d'accedir a les dades mitjançant `request.POST`, s'hi accedeix amb `form.cleaned_data` que retorna les dades validades i convertides a tipus de Python.

Seguint l'exemple anterior:

```
if form.is_valid():
    name = form.cleaned_data['name']
    message = form.cleaned_data['message']
    sender = form.cleaned_data['sender']
    cc_myself = form.cleaned_data['cc_myself']

    recipients = ['info@example.com']
    if cc_myself:
        recipients.append(sender)

    from django.core.mail import send_mail
    send_mail(subject, message, sender, recipients)
    return HttpResponseRedirect('/thanks/')
```

4.7.3.2 Formularis a les plantilles

Una de les tasques dels formularis és representar-se en HTML. Per a fer-ho, es passa una instància del formulari a la plantilla i allí s'utilitza la variable:

```
<form action="/contact" method="POST">
    {{ form.as_p }}
    <input type="submit" value="Submit" />
</form>
```

L'etiqueta `form.as_p` mostrarà cada camp del formulari en un paràgraf. Els formularis també es poden representar amb `form.as_ul` (per a representar-ho amb una llista) o `form.as_table` (per a representar-ho amb una taula). Aquests mètodes mostren tots els camps del formulari en el mateix ordre que estan definits dins de la classe.

El codi HTML que generen aquests mètodes no és del tot personalitzable i per solucionar-ho es poden representar els camps individualment accedint a l'objecte com un diccionari:

```
<form action="/contact" method="POST">
    <div class="field">
        {{ form.name.errors }}
        {{ form.name.label_tag }} {{ form.name }}
    </div>
    <div class="field">
        {{ form.message.errors }}
        {{ form.message.label_tag }} {{ form.message }}
    </div>
    <input type="submit" value="Submit" />
</form>
```

Cada camp del formulari es pot representar amb:

- `form.nom_del_camp`: representa el widget que li correspon.
- `form.nom_del_camp.errors`: representa els errors del camp, si n'hi ha.
- `form.nom_del_camp.label_tag`: representa el label pel camp.

4.7.3.3 Formularis des del Model

En desenvolupar una aplicació web basada en una base de dades, molts dels formularis estaran relacionats amb classes del Model. Per exemple, en el cas de l'aplicació que es desenvolupa en aquest treball, hi ha una classe `EntryComment` en el Model i es vol crear un formulari per a que els usuaris puguin enviar els seus comentaris. En aquest cas, seria redundant tornar a definir els camps en un formulari quan ja ho hem fet en una classe del Model. Per aquest motiu, Django proporciona una classe auxiliar que permet crear classes de formularis que es basin en models ja definits.

```
from django.forms import ModelForm

class CommentForm(ModelForm):
    class Meta:
        model = Comment
```

Els atributs de les classes del Model es transformen en camps del formulari i mantenen el seus arguments. En alguns casos no voldren que apareguin tots els camps sinó un subconjunt d'ells. Per fer-ho es poden utilitzar els atributs `fields` o `exclude`. El primer atribut serveix per especificar els camps que volem incloure i el segon permet especificar els que volem excloure:

```
from django.forms import ModelForm

class CommentForm(ModelForm):
    class Meta:
        model = Comment
        fields = ('name','comment')
```

O bé:

```
class CommentForm(ModelForm):
    class Meta:
        model = Comment
        exclude = ('email')
```

En tots dos casos, el formulari resultant és el mateix.

4.7.3.4 Validació

La validació d'un formulari habitualment es realitza quan es crida al mètode `is_valid`. Per defecte es valida que les dades enviades coincideixin amb els tipus esperats i que els camps obligatoris hi siguin presents. Si es vol personalitzar la validació es pot fer de vàries formes:

- Implementant el mètode `clean()` de la classe del formulari: aquest mètode permet fer qualsevol validació del formulari que necessiti accedir a varis camps. Per exemple: comparar dos camps de contrasenya per veure si coincideixen.
- Amb el mètode `clean_<nom_del_camp>()`: aquest mètode permet afegir regles de validació per un camp del formulari en concret.
- Implementant el mètode `clean()` d'una subclasse de `django.forms.Field`: aquest mètode és responsable de validar les dades de forma genèrica per aquest tipus de camp.

Qualsevol d'aquests mètodes pot llençar l'excepció `ValidationError` amb un missatge. Si és així, la variable `cleaned_data` estarà buida i el mètode `is_valid` retornarà False.

```
from django.forms import forms

class CommentForm(forms.ModelForm):
    class Meta:
        model = Comment

    def clean_email(self):
        invalid_emails = ['test@example.com','test@test.com']
        if self.cleaned_data['email'] in invalid_email:
            raise forms.ValidationError('This email is not valid.')
        return self.cleaned_data['email']
```

4.7.4 Internacionalització

L'objectiu de la internacionalització és permetre que una aplicació pugui oferir els seus continguts i funcionalitats en múltiples idiomes. Internament, Django està totalment internacionalitzat. Totes les cadenes de Django estan marcades per a poder-les traduir i conté més de 50 fitxers de localització. Aquest mateix sistema d'internacionalització està disponible per a usar-lo en les nostres aplicacions.

4.7.4.1 Cadenes de traducció

Per a especificar cadenes de text a traduir dins del codi Python s'utilitza la funció `ugettext()` que, per convenció, s'importa amb el nom `_` per reduir l'escriptura:

```
from django.utils.translation import ugettext as _

def blog_edit(user, blog, request):
    ...
    request.session['message'] = _('Blog settings saved')
```

Aquestes cadenes de text poden contenir també marques de posició:

```
from django.utils.translation import ugettext as _

def entry_photos_import(request, travel_key, entry_key):
    ...
    request.session['message'] = _('%s photos imported') % imported_photos
```

Si les cadenes a traduir es troben dins de les plantilles s'utilitzen dos etiquetes. L'etiqueta {% trans %} serveix per a traduir cadenes estàtiques:

```
<h3>{% trans "Get Your Free Travel Blog!" %}</h3>
```

Si es vol combinar text estàtic amb alguna variable s'ha d'utilitzar l'etiqueta {% blocktrans %}:

```
<h3>
    {% blocktrans %}
        {{ comments }} Comments on {{ title }}
    {% endblocktrans %}
<h3>
```

4.7.4.2 Fitxers d'idiomes

Un cop s'han marcat les cadenes de text a traduir, s'ha de crear un fitxer de missatges per a cada nou idioma. Un fitxer de missatges és un fitxer de text pla que conté totes les cadenes de text originals i la seva traducció per a aquest idioma. Aquests fitxers tenen extensió `.po`.

El sistema d'internacionalització de Django utilitza el mòdul `gettext` de Python. Emprant els scripts de control de Django es pot automatitzar el procés de creació i actualització d'aquests fitxers:

```
django-admin.py makemessages -l ca
```

En aquesta comanda, `ca` és el codi d'idioma que correspon al Català.

Aquest script recórre tot el projecte des de l'arrel i extreu aquelles cadenes que s'han marcat per a traduir i crea o actualitza el fitxer de missatges al directori `locale/LANG/LC_MESSAGES/`. En aquest exemple, el fitxer es trobarà a `locale/LANG/LC_MESSAGES/django.po`.

Per defecte, `django-admin.py` examina tots els fitxers de codi Python i fitxers .html. Si es vol canviar l'extensió es pot utilitzar l'opció `-e` o `--extension`:

```
django-admin.py makemessages -l ca -e txt
```

Els fitxers de missatges `.po` que s'hauran contindran les cadenes de text per a traduir-les:

```
#: travellog/admin/blog.py:31
msgid "Blog settings saved"
msgstr ""

#: travellog/admin/photos.py:86
#, python-format
msgid "%s photos imported"
msgstr ""

#: travellog/templates/main.html:7
msgid "Get Your Free Travel Blog!"
msgstr ""

#: travellog/templates/travelentry_detail.html:84
#, python-format
msgid "%(comments)s Comments on %(title)s"
msgstr ""
```

El camp `msgid` és l'identificador del missatge i no s'ha de modificar. La traducció s'ha de fer al camp `msgstr`.

Un cop creats els fitxers de missatges, aquests s'han de compilar per a que les traduccions estiguin disponibles. Per a fer-ho s'utilitza la comanda:

```
django-admin.py compilemessages
```

Aquesta comanda compila els fitxers `.po` i crea fitxers `.mo`, que són fitxers binaris optimitats per a `gettext`.

Tot aquest sistema (`gettext`, `.po` i `.mo`) és el de GNU gettext i és un estàndard en el programari lliure per a la internacionalització d'aplicacions.

4.7.5 Interfície d'administració

La creació d'interfícies d'administració és una de les tasques més repetitives en el desenvolupament web. Per molts no només és una tasca repetitiva sinó també avorrida. El procés és sempre el mateix: cal autenticar els usuaris, mostrar un llistat de registres, mostrar i validar formularis, etc, etc.

Una de les parts més potents de Django es la seva interfície d'administració automàtica que soluciona aquest problema i permet que el desenvolupador es dediqui a d'altres tasques.

4.7.5.1 Activar l'administració

L'administració de Django és totalment opcional. Per a activar-la cal editar el fitxer de configuració `settings.py`:

- S'ha d'afegir `django.contrib.admin` a la llista d'aplicacions instal·lades.
- És necessari que les aplicacions `django.contrib.auth`, `django.contrib.contenttypes` i `django.contrib.sessions` també estiguin instal·lades. L'administració de Django necessita aquestes aplicacions per a funcionar.

- Les classes, `django.middleware.common.CommonMiddleware`, `django.middleware.SessionMiddleware` i `django.contrib.auth.middleware.AuthenticationMiddleware` s'han d'incloure a `MIDDLEWARE_CLASSES`.

Executant la comanda `python manage.py syncdb` es creen les taules necessàries a la base de dades per utilitzar l'administració: usuaris, grups, rols...

L'administració de Django s'ha de mapejar al fitxer de URLs per a que sigui accessible:

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

urlpatterns = patterns('',
    # ...
    (r'^admin/', include(admin.site.urls)),
    # ...
)
```

4.7.5.2 Administrant els models

Per a que un model es pugui administrar, dins del directori de l'aplicació s'ha de crear un nou fitxer `admin.py` i registrar els models que es vulguin administrar:

```
from django.contrib import admin
from models import Travel

admin.site.register(Travel)
```

Amb això Django ja és capaç de crear un formulari que representi aquest model. No obstant, l'administració és molt més potent i permet editar els camps visibles, camps per a cercar, filtres, etc. Per a fer-ho s'ha de crear una classe nova que representa l'administració d'aquest model. Aquesta classe hereta de `admin.ModelAdmin`:

```
class TravelAdmin(admin.ModelAdmin):
    list_display = ['title', 'start_date', 'published']
    fields = ['title','start_date','end_date','published','text','slug']
    search_fields = ['title','text']
    ordering = '-start_date'

admin.site.register(Travel, TravelAdmin)
```

El codi anterior també registra la classe `Travel` per administrar-la però amb unes opcions diferents:

- `list_display`: indica els camps que es mostren al llistat de registres.
- `fields`: indica els camps que són editables.
- `search_fields`: indica els camps sobre els quals es pot cercar.
- `ordering`: indica l'ordre els registres en el llistat. En aquest cas, s'ordenaran pel camp `start_date` de forma descendent.

Aquestes opcions són només algunes de les més bàsiques. La resta d'opcions i configuracions disponibles es pot trobar a la documentació de Django [5].

4.8 Desplegament

Django inclou un servidor web simple per a realitzar proves durant el desenvolupament. Per a l'etapa de producció, es recomana l'ús d'Apache 2 amb els mòduls mod_python o mod_wsgi, tot i que Django suporta l'especificació WSGI i es pot executar en altres servidors.

WSGI és una especificació de Python que defineix una interfície entre servidors web i aplicacions. En certa manera, és equivalent a l'especificació dels servlets en Java.

Pel que fa a bases de dades, Django suporta PostreSQL, MySQL, Oracle i SQLite 3. Si s'utilitza Python 2.5 o superior, aquest ja inclou SQLite i no cal fer cap tipus d'instal·lació addicional.

En el cas d'aquest treball, Django s'executa sobre Google App Engine i s'utilitza la seva base de dades, la Datastore.

[5] http://docs.djangoproject.com/en/dev/ref/contrib/admin/

Capítol 5

Google App Engine

5.1 Introducció

Google App Engine (GAE) és una plataforma per a desenvolupar i hostatjar aplicacions web als servidors de Google. App Engine es basa en altres tecnologies internes de Google com BigTable i Google File System, tecnologies distribuïdes i escalables. D'aquesta manera, els desenvolupadors no s'han de preocupar dels sistemes, servidors o escalabilitat i només s'han de dedicar a la seva pròpia aplicació.

Avui en dia es suporten dos llenguatge de programació, Python i Java; tot i que Google ha dit que en un futur planeja suportar altres llenguatges. En aquest treball només s'ha estudiat la versió en Python ja que era l'únic llenguatge disponible en el principi. Qualsevol framework en Python que suporti WSGI es pot utilitzar a les aplicacions, a l'igual que altres llibreries escrites en Python.

A l'abril del 2008 és va publicar la primera versió beta amb un nombre limitat de comptes. Actualment qualsevol usuari de Google pot utilitzar aquest servei amb uns límits i quotes.

5.2 Limitacions i quotes

Les aplicacions s'executen en un entorn segur que proporciona accés limitat al sistema operatiu. Aquestes limitacions permeten que les peticions web es distribueixin en varis servidors i aquests augmentin i disminueixin segons la demanda de tràfic.

Algunes d'aquestes limitacions són:

- Una aplicació només pot accedir a altres serveis d'internet a través de les API de correu electrònic i extracció de URL que proporciona App Engine.
- Una aplicació no pot escriure en el sistema d'arxius i només es poden llegir els arxius pujats amb el codi de l'aplicació. Per emmagatzemar fitxers en temps d'execució s'ha d'utilitzar la base de dades.
- El codi de l'aplicació només es pot executar com a resposta a una sol·licitud web i ha de retornar una resposta en uns pocs segons. No es poden generar altres processos ni executar codi un cop s'ha enviat la resposta.

L'entorn d'App Engine inclou la biblioteca estàndard de Python tot i que alguns mòduls que violen les restriccions anteriors s'han eliminat i operacions com escriure fitxers o obrir sockets no es permeten.

A part d'aquestes limitacions en l'entorn de Python, les aplicacions poden consumir recursos gratuïtament fins a uns màxims. La majoria de les quotes de l'aplicació es restableixen cada 24 hores, excepte algunes que són fixes per tal de mantenir la integritat del sistema.

Algunes de les quotes de l'aplicació són les següents:

- Peticions i computació:
  - Peticions: 1.300.000
  - Temps de CPU: 6,5 hores.
  - Transferència entrant i sortint: 1 GB / 1 GB.
- Datastore:
  - Crides a la datastore: 10.000.000
  - Dades emmagatzemades: 1 GB.
- Email:
  - Crides a l'API: 7.000
  - Emails enviats a usuaris: 2.000
- URL Fetch:
  - Crides a l'API: 657.000
  - Dades enviades i rebudes: 4 GB / 4 GB.
- Manipulació d'imatges:
  - Crides a l'API: 864.000
  - Transformacions executades: 2.500.000
- Memcache:
  - Crides a l'API: 8.600.000
  - Dades enviades i rebudes: 10 GB / 50 GB

Si un desenvolupador vol que la seva aplicació creixi per sobre d'aquestes quotes poden habilitar un sistema de pagament. Aquest sistema permet comprar més recursos, establint el pressupost màxim diari.

5.3 Datastore

La Datastore proporciona un sistema d'emmagatzematge de dades transaccional i un motor de consultes en una API senzilla. La Datastore funciona sobre l'estructura robusta i escalable de Google i està dissenyada pensant en les aplicacions web.

5.3.1 Entitats i models

Un objecte de dades a la Datastore s'anomena entitat. Una entitat té una o més propietats, valors amb nom d'un dels tipus de dades suportats. Les entitats no segueixen cap esquema: dos entitats del mateix model no estan obligades a tenir les mateixes propietats o el mateix tipus de valors per a una propietat.

Cada entitat té una clau que la identifíca inequívocament. Les claus les genera automàticament la Datastore a partir del tipus de l'entitat i un nombre ID únic. La clau d'una entitat també la pot especificar la pròpia aplicació.

5.3.1.1 La classe `Model`

Una aplicació defineix els tipus de dades amb models. La idea és molt similar als models de Django: un model és una classe Python que hereta de la classe Model i defineix una o més propietats. Les propietats del model es defineixen emprant atributs de classe, on cada atribut és una instància d'una subclasse de la classe Property. Una instància d'una propietat manté la seva configuració: si la propietat és obligatoria o no, el seu valor per defecte, etc.

```
from google.appengine.ext import db

class Blog(db.Model):
    first_name = db.StringProperty(_('First Name'),required = True)
    last_name = db.StringProperty(_('Last Name'),required = True)
    user = db.UserProperty(required = False)
    url = db.StringProperty(_('Blog Url'), required = True)
    title = db.StringProperty(_('Blog Title'),required = True,
                              default='Your Blog Title')
    register_date = db.DateTimeProperty(auto_now_add = 1)
```

Una aplicació pot crear noves entitats d'algun dels tipus definits creant una nova instància. Les propietats es poden assignar en el mètode constructor o com a atributs després de crear la instància:

```
from google.appengine.api import users

blog = Blog(first_name='Nom', last_name='Cognoms', url='url', title='Títol')
blog.user = users.get_current_user()
```

La classe Model utilitza instàncies de Property per a validar el valor assignats a cada atribut. La validació dels valors es produeix quan la instància del model es crea per primera vegada i quan s'assigna un nou valor a un atribut de la instància. D'aquesta manera s'assegura que una propietat mai tindrà un valor invàlid.

Com que la validació es produeix quan la instància és creada, qualsevol propietat que estigui configurada com a obligatòria (required = True) haurà de ser inicialitzada en el constructor. En l'exemple anterior, first_name, last_name, url i title són propietats obligatòries que cal especificar en el constructor. La propietat user es pot assignar un cop la instància està creada.

5.3.1.2 Models `Expando`

Les classes definides amb la classe `Model` tenen unes propietats fixes que cada instància de la classe ha de tenir. Aquesta és una manera de modelar les dades, però a vegades pot ser útil tenir entitas que tenen propietats diferents a les altres entitats del mateix tipus. La datastore no obliga a que dues entitats del mateix tipus tinguin les mateixes propietats i per a definir aquest tipus d'entitats existeixen els models "expando".

Aquests models hereten de la classe `Expando` i tenen els mateixos mètodes que la classe `Model` bàsica. A part de les propietats fixes en tenen d'altres de dinàmiques. Quan s'assigna un valor a un atribut d'una instància d'un model expando, la datastore crea una propietat dinàmica amb el nom de l'atribut. Aquestes propietats s'anomenen dinàmiques, mentre que les propietats definides amb la classe `Property` són propieatats fixes.

El següent exemple mostra la diferència entre les propietats fixes i les dinàmiques:

```
from google.appengine.ext import db

class Person(db.Expando):
    first_name = db.StringProperty(required=True)
    last_name = db.StringProperty(required=True)
    hobbies = db.StringListProperty()

# Propietats fixes
p = Person(first_name='Albert', last_name='Johnson')
p.hobbies = ['chess', 'travel'] # és una propietat fixa pero no obligatòria

# Propietats dinàmiques
p.chess_rating = 1350
p.travel_countries_visited = ['Spain', 'Italy', 'USA']

# Propietats fixes
p2 = Person(firt_name='John', last_name='Doe')

# Propietats dinàmiques
p2.color = 'blue'
```

Ja que les propietats dinàmiques no tenen definicions de propietats del model, no es poden validar. No obstant, totes les propietats dinàmiques només poden tenir valors dels tipus bàsics de la datastore o `None`.

5.3.1.3 Models polimòrfics

La datastore inclou una altra classe per a modelar l'herència de classes. Aquests models s'anomenen "polimòrfics" ja que les instàncies d'una classe també apareixen com a resultats d'una consulta a la seva classe mare.

El següent exemple defineix la classe base `Contact` i dos subclasses `PersonContact` i `CompanyContact`:

```
from google.appengine.ext import db
from google.appengine.ext.db import polymodel
```

```
class Contact(polymodel.PolyModel):
    phone_number = db.PhoneNumberProperty(required=True)
    address = db.PostalAddressProperty(required=True)

class PersonContact(Contact):
    first_name = db.StringProperty(required=True)
    last_name = db.StringProperty(required=True)
    mobile_number = db.PhoneNumberProperty()

class CompanyContact(Contact):
    name = db.StringProperty(required=True)
    fax_number = db.PhoneNumberProperty()
```

Aquest model s'assegura que totes les entitats de `PersonContact` i `CompanyContact` tenen les propieats `phone_number` i `address`. Només les entitats de `PersonContact` tindran les propietats `first_name`, `last_name` i `mobile_number`. Igualment, només les entitats de `CompanyContact` tindran les propietats `name` i `fax_number`.

Les subclasses s'instancien com qualsevol altra classe del model:

```
p = PersonContact(phone_number='1-206-555-9234',
            address='123 First Ave., Seattle, WA, 98101',
            first_name='Alfred',
            last_name='Smith',
            mobile_number='1-206-555-0117')
p.put()

c = CompanyContact(phone_number='1-503-555-9123',
             address='P.O. Box 98765, Salem, OR, 97301',
             name='Data Solutions, LLC',
             fax_number='1-503-555-6622')
c.put()
```

Els models polimòrfics permeten que una consulta a la classe `Contact` retorni també entitats de `PersonContact` i `CompanyContact`. En canvi, una consulta a la classe `CompanyContact` només retornaria entitats d'aquesta classe.

5.3.2 Claus i grups d'entitats

Cada entitat de la datastore conté una clau única que la identifica i la diferencia de la resta d'entitats. La clau d'una entitat està composada per varis elements: una ruta (`path`) que descriu la relació entre aquesta entitat i una altra, el tipus d'entitat (`kind`) i un nom (`name`) assignat per l'aplicació o bé un identificador (`id`) numèric assignat automàticament per la datastore.

5.3.2.1 Identificadors

Cada entitat disposa d'un identificador que pot ser especificat per l'aplicació o bé generat automàticament per la datastore. Si l'aplicació vol assignar l'identificador, s'utilitza l'argument `key_name` en el constructor de l'entitat:

```
from google.appengine.ext import db
```

```
class TestModel(db.Model):
    title = db.StringProperty()

test = TestModel(key_name='key123', title='My title')
test.put() # L'entitat tindra un identificador assignat manualment
```

Si no s'especifica l'argument `key_name`, la datastore assignarà un identificador automàticament a l'entitat:

```
test2 = TestModel(title='My title')
test2.put() # La datastore assignara un identificador a l'entitat
```

Un cop creada una entitat, el seu identificador no es pot canviar.

5.3.2.2 Grups d'entitats i rutes

Cada entitat pertany a un grup d'entitats, un conjunt d'una o vàries entitats que es poden manipular en la mateixa transacció. Les relacions entre entitats del mateix grup indiquen a App Engine que emmagatzemi vàries entitats en la mateixa part de la seva infraestructura distribuïda.

Quan es crea una nova entitat, es pot assignar una altra entitat com a entitat principal d'aquesta nova. Al assignar una entitat principal a una nova entitat, la nova entitat es situa en el mateix grup d'entitats que la principal. Una entitat sense una entitat principal és una entitat arrel. La ruta d'una entitat és el camí que hi ha des d'ella fins a una entitat arrel.

L'entitat principal d'una entitat es defineix quan aquesta es crea i no es pot modificar posteriorment. Per a assignar una entitat principal a una de nova s'utilitza l'argument `parent` en el seu constructor:

```
from google.appengine.ext import db

class Blog(db.Model):
    ...

class Travel(db.Model):
    ...

blog = Blog()
blog.put()
travel = Travel(parent=blog)
travel.put()
```

Totes les entitats que tinguin la mateixa entitat principal pertanyen al mateix grup d'entitats. Totes les entitats d'un mateix grup s'emmagatzemen en el mateix node de la datastore. Una transacció només es pot aplicar sobre un grup d'entitats.

Els grups d'entitats només s'han d'utilitzar quan siguin necessàries les transaccions. Per a mapejar relacions entre entitats s'han d'utilitzar les claus i la propietat `ReferenceProperty`.

5.3.3 Propietats i tipus

La datastore admet un conjunt de tipus de valors per a les propietats de cada entitat, incloent nombres enters, cadenes de text, nombres decimals, dates, cadenes de bytes (blobs) i diversos tipus de dades de Google (GData). Cada tipus de dades de la datastore té la seva classe `Property` corresponent, disponible al mòdul `google.appengine.ext.db`.

La classe `Property` és la superclasse que permet definir propietats per a tots els models. Una classe `Property` defineix el tipus de valor d'una propietat, la forma en que es validen els valors i la forma en que s'emmagatzemen els valors a la datastore.

La següent taula descriu algunes de les subclasses de `Property` disponibles i el seu tipus corresponent.

Subclasse de `Property`	**Tipus**
StringProperty	str o unicode
BooleanProperty	bool
IntegerProperty	int o long
FloatProperty	float
DateTimeProperty	datetime.datetime
ListProperty	llista d'un dels tipus suportats
ReferenceProperty	db.Key
UserProperty	users.User
BlobProperty	db.Blob
TextProperty	db.Text
EmailProperty	db.Email
GeoPtProperty	db.GeoPt
PhoneNumberProperty	db.PhoneNumber

Taula 5.1: Propietats de la datastore

La llista completa de propietats està disponible a la documentació de la datastore de Google App Engine [1]

Cada propietat de la datastore té el seu tipus associat. Alguns són tipus bàsics de Python (str, bool, int, float...) i d'altres són tipus propis de Google, la major part disponibles al mòdul `google.appengine.ext.db`.

El següent exemple mostra l'ús de les propietats:

```
from google.appengine.ext import db

class TravelEntry(db.Model):
    ...
    title = db.StringProperty(required=True)
    text = db.TextProperty(required=True)
    point = db.GeoPtProperty()
```

[1] http://code.google.com/intl/ca/appengine/docs/python/datastore/typesandpropertyclasses.html

```
        ...

entry = TravelEntry(title='My entry', text=db.Text('Entry text'))
entry.point(db.GeoPt(41.6141518,0.6257825))
```

En aquest exemple, la diferència entre `StringProperty` i `TextProperty` no és unicament la seva llargada. La propietat `StringProperty` representa cadenes de text curtes, d'una llargada màxima de 500 bytes. La propietat `TextProperty` representa cadenes de text més llarges, de fins a 1 MB. Les cadenes de text curtes s'indexen i es poden utilitzar per a filtrar i ordenar resultats, mentre que les cadenes llargues no s'indexen i no es poden utilitzar d'aquesta manera.

Pel que fa a la propietat `GeoPtProperty`, representa un punt en el mapa i els seus paràmetres en el constructor són les coordenades del punt en aquest ordre: latitud, longitud.

Si es vol, es poden definir nous tipus de propietats. Per a fer-ho, cal crear una nova classe que hereti de `Property` i que redefineixi els següents mètodes `get_value_for_datastore()` i `make_value_from_datastore`. Aquests mètodes indiquen com s'han de convertir els tipus de Python a la datastore i viceversa. Opcionament, es poden redefinir els mètodes `validate()` i `empty()` que indiquen com s'han de validar els valors i en quins casos es considera un valor buit.

5.3.4 Crear, obtenir i eliminar dades

5.3.4.1 Crear i actualitzar dades

Per a crear una nova instància d'una classe del Model s'utilitza el seu constructor:

```
from google.appengine.ext import db

entry = TravelEntry(title='My entry', text=db.Text('Entry text'))
```

La nova instància no estarà creada a la datastore fins que es crida al mètode `put()` de la instància o bé es passa la instància a la funció `db.put()`. El resultat és el mateix:

```
from google.appengine.ext import db

entry.put()

db.put(entry)
```

Si la instància ja existeix a la datastore, el mètode `put()` actualitza l'entitat.

5.3.4.2 Obtenir entitats amb una consulta

La datastore pot executar consultes i retornar entitats d'una de les classes del Model. Les consultes poden filtrar els resultats a partir d'alguna de les seves propietats i també els poden ordenar. L'API de la datastore proporciona dues interfícies per a realitzar consultes: `Query`, una interfície que prepara consultes fent servir mètodes d'un objecte de consulta, i `GqlQuery`, una interfície que usa un llenguatge pseudo-SQL anomenat GQL (Google Query Language).

La classe `Query`

El mètode `all()` d'una classe del Model retorna un objecte `Query` que representa una consulta a totes les entitats d'aquell tipus. Un objecte `Query` també es pot obtenir cridant el seu constructor i passant-li com a argument la classe que es vol consultar:

```
from google.appengine.ext import db

class Travel(db.Model):
    ...
    title = db.StringProperty(required=True)
    text = db.TextProperty(required=True)
    published = db.BooleanProperty(default=False)
    start_date = db.DateProperty(required=True)
    ...

query = db.Query(Travel)

query = Travel.all()
```

A partir de l'objecte `Query`, l'aplicació preapara la consulta cridant als mètodes `filter()`, `order()` i `ancestor()`:

```
query = Travel.all() # S'obté l'objecte Query

query.filter('published =', True) # Es filtren els viatges publicats
# S'ordenen les entitats pel camp start_date de forma descendent
query.order('-start_date')

# Aquests mètodes es poden encadenar en una sola línia
query = Travel.all().filter('published =', True).order('-start_date')
```

La classe `GqlQuery`

La classe `GqlQuery` permet fer consultes a la datastore amb un llenguatge semblant a SQL. La consulta es representa amb una cadena de text on s'especifica la classe a la que s'ha d'aplicar, els filtres, criteris d'ordenació, etc. Els paràmetres a la consulta es poden especificar com a arguments posicionals, arguments amb nom o com a literals dins de la cadena:

```
from google.appengine.ext import db

# Paràmetres amb arguments posicionals
query = db.GqlQuery("SELECT * FROM Travel WHERE published = :1 "
                    "ORDER BY start_date DESC",
                    True)

# Paràmetres amb arguments amb nom
query = db.GqlQuery("SELECT * FROM Travel WHERE published = :published "
                    "ORDER BY start_date DESC",
                    published=True)

# Paràmetres dins la cadena
query = db.GqlQuery("SELECT * FROM Travel WHERE published = True "
                    "ORDER BY start_date DESC")
```

En aquest exemple, la consulta resultant en els tres casos és la mateixa.

Les classes del Model tenen el mètode `gql()` que també prepara un objecte `GqlQuery` a partir d'una cadena de text. En aquest cas, l'expressió "SELECT * FROM Model" s'omet ja que està implícita en la pròpia classe:

```
query = Travel.gql("WHERE published = True "
                   "ORDER BY start_date DESC")
```

Tot i que la sintaxis de GQL és molt semblant a SQL i la majoria d'operadors estan permesos, les característiques del llenguatge són molt diferents i hi ha certes diferències i limitacions respecte SQL. Per exemple:

- No existeixen els JOINS.
- No es poden aplicar funcions aritmètiques: COUNT, SUM, MAX...
- No es permet l'ús de DISTINCT o GROUP BY.

5.3.4.3 Executar les consultes i accedir als resultats

Els objectes `Query` i `GqlQuery` només preparen les consultes, no les executen. Una consulta s'executa quan l'aplicació accedeix als seus resultats. Hi ha tres maneres d'accedir als resultats d'una consulta:

- El mètode `fetch()` rep com a paràmetre el nombre màxim de resultats que es vol obtenir (límit) i el nombre de resultats que s'han de s'ignorar (desviació). Aquest mètode executa la consulta i obté els resultats fins que arriba al límit o bé no n'hi ha més. Quan els resultats s'han carregat en memòria, s'ignoren els primers resultats si s'ha especificat una desviació.

  ```
  query = Travel.all()
  results = query.fetch(10)
  for travel in results:
      print 'Title :', travel.title
  ```

- El mètode `get()` executa la consulta i retorna únicament el primer resultat o `None` si no n'hi ha cap. L'ús d'aquest mètode implica un límit d'1.

  ```
  query = Travel.all()
  first_travel = query.get()
  print 'Title :', travel.title
  ```

- Els objectes `Query` i `GqlQuery` també es poden utilitzar com a iteradors. En aquests casos, s'executa la consulta sense límit ni desviació. Els resultats es carreguen en memòria i es retorna un iterador per a aquests resultats:

  ```
  query = Travel.all()
  for travel in query:
      print 'Title :', travel.title
  ```

Cal tenir en compte que la datastore retorna un màxim de 1000 resultats com a resposta a una consulta, independentment del límit o de la desviació.

5.3.4.4 Obtenir entitats mitjançant una clau

Cada entitat emmagatzemada a la datastore té una clau única. Els valors de les claus es representen en forma d'instàncies de la classe `Key`. Quan s'emmagatzema una entitat a la datastore, els mètode `put()` i la funció `db.put()` retornen la seva clau. Quan una entitat ja existeix a la datastore, el mètode `key()` retorna la seva clau.

Obtenir una entitat a partir de la seva clau és la consulta més ràpida que es pot fer a la datastore. Internament, la datastore és una taula hash que associa claus amb valors, i per això cada claus és única en tota l'aplicació.

Per a obtenir una entitat a partir de la seva clau es pot utilitzar el mètode `get()` d'un model o bé la funció `db.get()`. En tots dos casos s'ha de passar com a paràmetre la clau o una llista de claus:

```
from google.appengine.ext import db

travel = Travel(title='My travel', text=db.Text('Travel description...'))
key = travel.put() # En guardar una entitat es retorna la clau

travel2 = Travel.get(key) # Obtenir una entitat a partir de la seva clau
travel3 = db.get(key) # Obtenir una entitat a partir de la seva clau
```

L'entitat que s'obté en tots dos casos és la mateixa. La diferència entre les dos opcions és que el mètode `Travel.get()` assegura que l'entitat retornada és de la classe `Travel` i si no ho és llença una excepció. En canvi, la funció `db.get()` retorna entitats de qualsevol classe.

5.3.4.5 Esborrar entitats

Les aplicacions poden eliminar dades de la datastore a partir d'una instància d'un Model o amb la seva clau. El mètode `delete()` d'una instància d'un model esborra l'entitat de la datastore. En canvi, la funció `db.delete()` esborra una o vàries entitats a partir de les seves claus o amb les seves instàncies:

```
from google.appengine.ext import db

query = Travel.all()
for travel in query:
    travel.delete() # Esborra una per una les entitats

query = db.GqlQuery("SELECT * FROM Travel WHERE published = True")
results = query.fetch(10)
db.delete(results) # Esborra de cop les 10 primeres entitats
```

Si s'esborra una entitat, no es modifica cap referència que pugui tenir aquella entitat. La datastore no esborra entitats en cascada, és l'aplicació la que s'ha d'encarregar d'esborrar o modificar les entitats relacionades.

5.3.5 Referències i relacions

La datastore també permet representar relacions entre classes del Model, d'una manera força semblant a com ho feia Django.

5.3.5.1 One-to-Many

Les relacions One-to-Many es modelen amb la propietat `ReferenceProperty`. Aquesta propietat guarda la clau de l'entitat relacionada:

```
from google.appengine.ext import db

class Travel(db.Model):
    ...
    title = db.StringProperty(required=True)
    text = db.TextProperty(required=True)
    ...

class TravelEntry(db.Model):
    travel = db.ReferenceProperty(Travel, collection_name='entries')
    title = db.StringProperty(required=True)
    text = db.TextProperty(required=True)
    point = db.GeoPtProperty()
```

Per a assignar una referència es pot utilitzar una instància d'un model o bé la seva clau:

```
travel = Travel(title='My First Travel', text=db.Text('Intro text...'))
travel.put()

first_entry = TravelEntry(title='My First Entry', text=db.Text('Text...'))
first_entry.travel = travel # S'assigna la referència
first_entry.put()

second_entry = TravelEntry(title='My Second Entry', text=db.Text('Text...'))
second_entry.travel = travel.key() # S'assigna la referència
second_entry.put()
```

En aquest exemple, la referència és la mateixa en les dues instàncies de `TravelEntry` i sempre es guarda la clau de l'objecte relacionat.

Quan un model té una propietat `ReferenceProperty` que fa referència a un altre model, cada entitat del model referenciat automàticament rep una propietat que permet obtenir totes les entitats del primer model que fan referència a ella. Aquesta propietat es pot especificar amb l'argument `collection_name`, tal com s'ha fet en la definció de la classe `TravelEntry`.

D'aquesta manera, les entitats de `Travel` tenen una nova propietat `entries` que permet consultar totes les entitats relacionades:

```
for entry in travel.entries:
    print entry.title
```

L'atribut `entries` és un atribut virtual que internament és una instància de la classe `Query`. Això vol dir que sobre aquest atribut es poden aplicar filtres, ordenar les dades, esborrar-les... Per exemple:

```
for entry in travel.entries.filter('title =', 'My First Entry'):
    print entry.title, entry.text
```

5.3.5.2 Many-to-Many

Les relacions Many-to-Many es modelen guardant una llista de claus en un dels costats de la relació. La millor manera d'entendre aquest tipus de relacions és amb un exemple amb contactes i grups:

```
from google.appengine.ext import db

class Group(db.Model):
    name = db.StringProperty()
    description = db.TextProperty()


class Contact(db.Model):
    ...
    groups = db.ListProperty(db.Key)      # Grups als que pertany
```

Les instàncies de la classe `Contact` tenen un atribut `groups` que és una llista en Python. Aquesta llista pot emmagatzemar objectes del tipus `db.Key`, claus dels objectes relacionats.

Per a afegir o eliminar grups d'un contacte només cal treballar amb els mètodes habituals d'una llista en Python:

```
friends = Group.gql("WHERE name = 'friends'").get()
mary = Contact.gql("WHERE name = 'Mary'").get()
if friends.key() not in mary.groups:
    mary.groups.append(friends.key())
    mary.put()
```

El codi de l'exemple anterior afegeix el contacte a un nou grup si aquest encara no hi pertany.

5.3.6 Consultes i índexos

Una consulta obté entitats de la datastore que cumpleixen un conjunt de condicions. Cada consulta especifica un tipus d'entitat, zero o més condicions basades en valors de les seves propietats i zero o més criteris d'ordenació. Quan s'executa una consulta, s'obtenen totes les entitats del tipus concret que cumpleixen totes les condicions especificades i s'ordenen segons els criteris d'ordenació.

Cada consulta a la datastore utilitza un índex, una taula que conté els resultats de la consulta ordenats d'una determinada manera. A mesura que l'aplicació realitza canvis a les entitats, la datastore actualitza els índexos amb els resultats correctes. Quan l'aplicació executa una consulta, la datastore extreu els resultats de l'índex corresponent. Si l'aplicació està en producció i executa una consulta que no té un índex, la consulta fallarà.

Una aplicació de Google App Engine defineix els seus índexos en un fitxer de configuració que s'anomena `index.yaml`. El servidor web de l'entorn de desenvolupament s'encarrega de generar automàticament aquest fitxer a mesura que troba consultes que no tenen un índex associat. Els índexos es poden configurar manualment editant aquest fitxer.

Prenem com a exemple la següent consulta:

```
Travel.all().filter('blog = ', blog).filter('published = ', True)
    .order('-start_date')

# Equivalencia en GQL
Travel.gql("WHERE blog = :blog AND published = True
           ORDER BY start_date DESC", blog=blog)
```

Si l'aplicació executa aquesta consulta el seu índex serà així:

```
- kind: Travel
   properties:
  - name: blog
  - name: published
  - name: start_date
     direction: desc
```

Un índex conté com a mínim els següents valors:

- `kind`: el tipus de l'entitat de la consulta. És el nom de la classe del Model i és obligatori.
- `properties`: una llista de les propietats que s'inclouran com a columnes de l'índex. Cada element de la llista ha d'indicar el nom de la propietat i opcionalment la direció d'ordenació.

En aquest cas, l'índex s'aplica sobre la classe `Travel`, utilitza les propietats `blog` i `published` per a filtrar les dades i utilitza la propietat `start_date` per a ordenar-les.

5.3.7 Transaccions

Una transacció és un conjunt d'operacions a una base de dades que es resolen correctament o incorrectament en la seva totalitat. Si la transacció és correcta, tots els canvis s'apliquen. Si la transacció no és correcta, no s'aplica cap dels canvis.

La datastore proporciona la funció `run_in_transaction()` que rep una funció com a paràmetre i executa les seves operacions dins de la mateixa transacció:

```
from google.appengine.ext import db

class Accumulator(db.Model):
    counter = db.IntegerProperty()

def increment_counter(key, amount):
    obj = db.get(key)
    obj.counter += amount
    obj.put()

q = db.GqlQuery("SELECT * FROM Accumulator")
acc = q.get()

# Executa la operacions de la funció dins d'una transacció
db.run_in_transaction(increment_counter, acc.key(), 5)
```

Les transaccions a la datastore tenen limitacions o restriccions. Això es degut a que la datastore és una base de dades distribuïda i fer transaccions entre diferents servidors comporta una dificultat afegida.

Totes les operacions de la datastore incloses en una mateixa transacció s'han de realitzar sobre entitats del mateix grup. Una transacció no pot fer consultes amb les classes `Query` o `GqlQuery`, i únicament es permeten els mètodes `put()`, `get()` i `delete()`.

5.4 APIs de serveis

Google App Engine proporciona varis serveis útils basats en la seva infraestructura. Les aplicacions poden accedir a aquests serveis utilitzant llibreries incloses dins del SDK.

5.4.1 Comptes de Google

Les aplicacions de Google App Engine poden utilitzar comptes de Google per a l'autenticació. La integració amb comptes de Google és totalment opcional i, si es vol, sempre es pot desenvolupar un sistema propi d'autenticació. No obstant, Google ofereix un sistema d'autenticació robust, madur i totalment fiable que automàticament obre la nostra aplicació a milions d'usuaris. D'aquesta manera, l'usuari pot utilitzar un compte ja existent per a auntenticar-se a la nostra aplicació.

L'ús de comptes de Google per a l'autenticació proporciona avantatges tant pels usuaris com pels desenvolupadors:

- **Desenvolupadors:** la gestió i autenticació d'usuaris és una de les tasques repetitives i delicades del desenvolupament web. Cal gestionar usuaris, encriptar les contrasenyes, permetre recuperar les contrasenyes, etc, etc. Tots aquests problemes es solucionen ràpidament amb l'ús de les comptes de Google.
- **Usuaris:** els usuaris no s'han de crear un compte exclusiu per a una web ni confiar-hi per a donar les seves dades. Utilitzen el seu compte de google per a totes les aplicacions, d'una forma semblant a OpenID.

Una aplicació pot detectar si l'usuari està autenticat o redigir-lo a una pàgina per a autenticar-se o registar-se. Quan un usuari està autenticat, l'aplicació únicament pot accedir al seu email, "nickname" i identificador. L'aplicació també pot saber si l'usuari autenticat és un administrador de l'aplicació, de manera que es poden crear àrees d'administració de forma senzilla.

5.4.1.1 Ús de l'API

Els següent exemple mostra com s'utilitza l'API d'usuaris:

```
from django.http import HttpResponse
from google.appengine.api import users

def user_view(request):
    user = users.get_current_user()
    if user:
        output = ("Welcome, %s! (<a href=\"%s\">sign out</a>)" %
```

```
            (user.nickname(), users.create_logout_url("/")))
    else:
        output = ("<a href=\"%s\">Sign in or register</a>." %
            users.create_login_url("/"))
    return HttpResponse("<html><body>%s</body></html>" % output)
```

Aquest codi conté tres parts importants:

- La funció `get_current_user()` retorna l'usuari si està autenticat o `None` si no ho està.
- Si l'usuari està autenticat es mostra un missatge personalitzat amb el seu "nickname" i un enllaç per a desconectar-se.
- Si l'usuari no està autenticat es mostra un enllaç per a que s'autentiqui.

Les funcions `users.create_login_url()` i `users.create_logout_url()` retornen URLs de Google per a autenticar-se o desconectar-se. La pàgina d'autenticació de Google informa a l'usuari que s'està autenticant per a la nostra aplicació. Un cop l'usuari s'ha autenticat o desconectat, es redirigeix a l'aplicació, a la URL que s'ha rebut com a paràmetre.

Una aplicació pot comprobar si l'usuari autenticat és administrador de l'aplicació. L'administrador pot accedir a la consola d'administració de l'aplicació. La funció `users.is_current_user_admin()` retorna `True` si l'usuari autenticat és administrador.

5.4.1.2 La classe `User`

Una instància de la classe `User` representa un usuari amb un compte de Google. Dos objectes de la classe `User` són comparables: si són iguals, representen el mateix usuari. A part d'obtenir una instància de l'usuari autenticat amb la funció `get_current_user()`, les instàncies també es poden construir a partir d'un email:

```
from google.appengine.api import users

user = users.User('example@gmail.com')
```

Si el constructor es crida amb un email que no correspon a un compte de Google vàlid, l'objecte es crearà però no correspondrà a un compte de Google real. Quan s'utilitza el servidor de desenvolupament en local, aquesta restricció no s'aplica i tots els objectes de la classe `User` representen comptes de Google simulats.

Una instància d'`User` té els següents mètodes:

- `nickname()`: retorna el "nickname" de l'usuari.
- `email()`: retorna l'email de l'usuari.
- `user_id()`: retorna l'identificador permanent de l'usuari. Aquest identificador no canvia si l'usuari crea un nou email associat al seu compte.

La datastore permet guardar instàncies de la classe `User` amb la propietat `UserProperty`:

```
from google.appengine.api import users

class Blog(db.Model):
    user = db.UserProperty(required=False)
    ...
```

5.4.2 Memcache

El servei de Memcache proporciona a l'aplicació una memòria d'alt rendiment accessible des de múltiples instàncies de l'aplicació. Aquest servei és útil per dades que no necessiten la persistència o l'entorn transaccional de la datastore. L'API de Memcache permet incrementar el rendiment de l'aplicació i reduir la càrrega de la datastore.

El patró típic d'ús d'una memòria cau és el següent:

- L'aplicació rep una petició d'un usuari que necessita algunes dades.
- L'aplicació consulta si les dades que necessita es troben a la memòria cau:
  - Si les dades es troben en la memòria cau, l'aplicació utilitza aquestes dades.
  - Si les dades no s'hi troben, l'aplicació consulta a la base de dades aquestes dades i les guarda en la memòria cau per a consultes posteriors.

El següent exemple mostra l'ús de l'API de Memcache:

```
def get_last_blogs(request):
    last_blogs = memcache.get('last_blogs')
    if not last_blogs:
        last_blogs = Blog.all().order('-register_date').fetch(5)
        memcache.set('last_blogs', last_blogs)
    ...
```

En aquest codi hi ha dos funcions relacionades amb aquesta API:

- La funció `memcache.get('last_blogs')` retorna el valor que conté la clau 'last_blogs' o `None` si no n'hi ha cap.
- La funció `memcache.set('last_blogs', last_blogs)` emmagatzema la variable `last_blogs` amb la clau 'last_blogs'. Si aquesta clau ja existeix, el valor es sobreescriu.

En aquest cas, l'API de Memcache s'utilitza en la vista que mostra els últims blogs registrats. Aquestes dades es mostren a la pàgina principal de l'aplicació i utilitzant Memcache s'evita que es consultin cada vegada. Les dades només es consulten a la base de dades si no es troben en la memòria cau. Cal tenir en compte que quan s'afegeix un nou blog cal esborrar o recalcular aquestes dades ja que sinó no seran correctes.

Les dades en Memcache es guarden igual que en un diccionari, amb una estructura clau : valor. Per defecte, les dades emmagatzemades es guarden el major temps possible i s'esborren quan es volen afegir noves dades i la memòria està plena. Quan això passa, s'esborren les dades menys usades recentment.

Una aplicació també pot especificar el temps d'expiració de les dades. Quan passa aquest temps, si les dades encara es troben en memòria, s'esborren. Tot i especificar aquest temps, les dades es poden haver esborrat anteriorment si falta memòria o si es produeix algun error en la plataforma.

L'API de Memcache té altres funcions per a esborrar dades, obtenir múltiples dades per a múltiples claus, esborrar tota la Memcache, obtenir estadístiques, etc.

5.4.3 URL Fetch

Les aplicacions de Google App Engine poden comunicar-se amb altres hosts i serveis mitjançant peticions HTTP o HTTPS. Per fer-ho, existeix el servei URL Fetch que utilitza la infraestructura de Google per a obtenir escalabilitat i eficiència.

Les llibreries estàndard de Python `urllib`, `urllib2` i `httplib` estan reescrites per a que internament utilitzin Url Fetch. Això permet la compatibilitat d'aplicacions i altres llibreries que les utilitzin sense haver de retocar el codi.

Es pot accedir a aquest servei fent servir l'API:

```
def get_country(point):
    from google.appengine.api import urlfetch
    url = 'http://maps.google.com/maps/geo?q=%s&output=csv&hl=en&key=%s'
            % (str(point), settings.GMAPS_KEY)
    try:
        result = urlfetch.fetch(url)
        if result.status_code == 200:
            return result.content.replace('"','').split(',')[-1]
        else:
            return None
    except Exception:
        return None
```

Aquest codi utilitza l'api d'URL Fetch per a obtenir el país que correspon a unes coordenades. Si la petició té èxit (`result.status_code == 200`), es retorna el país. Si es produeix algun error HTTP o qualsevol excepció, es retorna `None`.

El servei de URL Fetch suporta cinc mètòdes HTTP: `GET`, `POST`, `HEAD`, `PUT` i `DELETE`. La petició pot incloure capçaleres HTTP i contingut per a peticions `POST` o `PUT`. Per exemple, per a enviar dades a un formulari utilitzant el mètode `POST`:

```
import urllib

form_fields = {
  "first_name": "John",
  "last_name": "Doe",
  "email_address": "test@example.com"
```

```
}
form_data = urllib.urlencode(form_fields)
result = urlfetch.fetch(url=url,
                        payload=form_data,
                        method=urlfetch.POST,
                        headers={'Content-Type':
                                 'application/x-www-form-urlencoded'})
```

Aquest servei només permet els protocols HTTP i HTTPS i fent servir els ports estàndard, 80 i 443. Si la petició és HTTPS, les dades es transmeten encriptades a través de la xarxa i de la infraestructura de Google. Per defecte, el temps límit d'una petició és de 5 segons però es pot modificar fins a un màxim de 10 segons.

El servei URL Fetch retorna totes les dades de la resposta: codi de resposta, capçaleres i contingut. Per defecte, si la resposta excedeix un límit (1 MB), es trunca el seu contingut. En aquest cas, l'atribut `response_was_truncated` retorna `True`.

5.4.3.1 Peticions asíncrones

L'API en Python del servei URL Fetch suporta també peticions asíncrones. Amb una petició síncrona, l'API espera fins que el host remot retorna un resultat i llavors retorna el control a l'aplicació.

En canvi, amb una petició asíncrona, l'API crea un objecte RPC que representa una crida asíncrona i retorna inmedietament el control a l'aplicació. Així, l'aplicació pot realitzar altres tasques mentre s'executa la petició i quan necessita els resultats, pot cridar un mètode de l'objecte RPC per a obtenir-los.

En el següent exemple es pot veure com funciona una petició asíncrona:

```
from google.appengine.api import urlfetch

rpc = urlfetch.create_rpc()
urlfetch.make_fetch_call(rpc, "http://www.google.com/")

# ... realitzar altres tasques ...

try:
  result = rpc.get_result()
  if result.status_code == 200:
    text = result.content
    # ...
except urlfetch.DownloadError:
  # La petició ha fallat o s'ha produït un timeout
  # ...
```

En aquest exemple, l'objecte RPC es crea amb la funció `urlfetch.create_rpc()` i la petició comença amb `urlfetch.make_fetch_call()`. Quan es crida el mètode `get_result()` de l'objecte RPC, l'aplicació espera a que la petició acabi i li retorni un resultat.

A l'igual que en les peticions síncrones, les asíncrones també tenen un temps límit per defecte de 5 segons que es pot augmentar fins a 10 segons.

5.4.4 Email

Les aplicacions de Google App Engine poden enviar emails utilitzant el servei que proporciona la plataforma. L'API d'email de Google App Engine proporciona dos formes per a enviar emails: la funció `mail.send_mail()` i la classe `EmailMessage`.

La funció `mail.send_mail()` rep com a paràmetres tots els camps del missatge i l'envia:

```
from django.utils.translation import ugettext as _
from google.appengine.api import users, mail

def register(request):
    user = users.get_current_user()
    blog = get_object(Blog, 'user =', user)
    ...
    subject = _('Welcome to Blog Your Travel')
    body = _("""
            Welcome to Blog Your Travel
            This is your blog url:
            http://blogyourtravel.appspot.com%s
            """) % (blog.get_permalink())
    mail.send_mail('josepb@gmail.com', blog.user.email(), subject, body)
```

La classe `EmailMessage` proporciona la mateixa funcionalitat però utilitzant objectes. Els camps del missatge es poden passar a l'objecte en el seu constructor o bé utilitzant atributs de la instància. L'email s'envia amb el mètode `send()`:

```
from django.utils.translation import ugettext as _
from google.appengine.api import mail

def register(request):
    user = users.get_current_user()
    blog = get_object(Blog, 'user =', user)
    ...
    message = mail.EmailMessage(sender='josepb@gmail.com',
                            subject=_('Welcome to Blog Your Travel'))
    message.to = blog.user.email()
    message.body = _("""Welcome to Blog Your Travel
                    This is your blog url:
                    http://blogyourtravel.appspot.com%s
                    """) % (blog.get_permalink())
    message.send()
```

Es poden utilitzar altres camps dels emails com ara `cc`, `bcc`, `reply_to` o fins i tot `html` per a enviar contingut HTML.

L'enviament d'emails és asíncron: tan la funció `mail.send_mail()` com el mètode `send()` de la classe `EmailMessage` envien el missatge al servei d'email de la plataforma i retornen el control a l'aplicació. El servei encua el missatge i més endavant l'envia.

5.4.4.1 Fitxers adjunts

Les aplicacions de Google App Engine poden enviar fitxers adjunts en els emails. El camp `attachments` accepta una llista de fitxers adjunts, on cada fitxer adjunt és una tupla amb el nom del fitxer i el seu contingut:

```
from google.appengine.api import mail
from google.appengine.ext import db

class DocFile(db.Model):
    doc_name = StringProperty()
    doc_file = BlobProperty()

q = db.GqlQuery("SELECT * FROM DocFile WHERE doc_name = :1", myname)
doc = q.fetch(1).get()
if doc:
    mail.send_mail(sender='support@example.com',
                   to='Albert Johnson <Albert.Johnson@example.com>',
                   subject='The doc you requested',
                   body='Attached is the document file you requested',
                   attachments=[(doc.doc_name, doc.doc_file)])
```

Per temes de seguretat, només es permeten enviar alguns tipus de fitxers: bmp, css, csv, gif, html, jpeg, pdf, txt...

5.4.5 Imatges

Google App Engine proporciona una llibreria per a manipular imatges. El servei d'imatges permet redimensionar-les, rotar-les, voltejar-les, retallar-les i fins i tot millorar-les utilitzant un algorisme predefinit. L'API també pot obtenir informació de la imatge: format, amplada, llargada...

L'API d'imatges accepta els formats JPEG, PNG, GIF, BMP, TIFF i ICO però les imatges que produeix després de les transformacions únicament són JPG o PNG. La mida màxima de les imatges és 1 MB.

En el següent exemple, l'API d'imatges s'utilitza per a crear una miniatura d'una imatge pujada per l'usuari:

```
from google.appengine.api import images
from travellog.util import getImageInfo

def blog_about(user, blog, request):
    ...
    try:
        uploaded_image = request.FILES['photo']
        image_data = uploaded_image.read()
        content_type, width, height = getImageInfo(image_data)
        if width >= height and width>250:
            image_thumb = images.resize(image_data, width=250)
        elif width < height and height>250:
            image_thumb = images.resize(image_data, height=250)
        else:
            image_thumb = None
        image = Image(image=db.Blob(image_data), thumb=image_thumb)
        image.put()
```

```
    except Exception:
        pass
```

A l'igual que en l'API d'email, les transformacions no només es poden fer mitjançant funcions del paquet `google.appengine.api.images` sinó també utilitzant la classe `Image` del mateix paquet:

```
from google.appengine.api import images

def blog_about(user, blog, request):
    ...
    uploaded_image = request.FILES['photo']
    img = images.Image(uploaded_image.read())
    img.resize(width=250, height=150)
    img.im_feeling_lucky()
    thumbnail = img.execute_transforms(output_encoding=images.JPEG)
    ...
```

Utilitzant aquesta classe, es poden aplicar vàries transformacions que s'executen quan es crida el mètode `execute_transforms()`. El mètode `resize()` redimensiona la imatge a les mides que se li passen i el mètode `im_feeling_lucky()` millora la qualitat de la imatge aplicant un algorisme similar al que hi ha disponible a Picasa Web Albums.

Per a utilitzar l'API d'imatges en local és necessari tenir instal·lada la llibreria Python Imaging Library (PIL).

5.5 Configuració de l'aplicació

La configuració d'una aplicació de Google App Engine es descriu en un fitxer anomenat `app.yaml`, al directori arrel. En aquest fitxer es defineix l'entorn d'execució de Google App Engine, la versió de l'aplicació i el mapeig de les URLs de l'aplicació:

```
application: blogyourtravel
version: 1
runtime: python
api_version: 1

handlers:
- url: /
  script: home.py

- url: /index\.html
  script: home.py

- url: /static
  static_dir: static

- url: /user
   script: user.py
   login: required

- url: /admin/.*
  script: admin.py
  login: admin
```

```
- url: /.*
  script: not_found.py
```

Aquest fitxer ha de contenir els següents elements:

- `application`: és l'identificador de l'aplicació i ha de coincidir amb el nom de l'aplicació creada a http://appengine.google.com.
  L'aplicació estarà disponible a http://blogyourtravel.appspot.com
- `version`: Google App Engine conserva una copia de l'aplicació per a cada versió utilitzada. L'administrador pot canviar la versió utilitzada mitjançant la interfície d'administració i també provar versions de prova abans de fer-les públiques.
- `runtime`: és el nom de l'entorn d'execucuió utilitzat per l'aplicació. Actualment només hi ha dos entorns: python o java.
- `api_version`: la versió de l'API de l'entorn d'execució utilitzat per l'aplicació. Actualment només hi ha una versió de l'API en Python.
- `handlers`: conté una llista d'URLs i informació de com s'han de processar. Per a processar les URLs, App Engine pot executar codi de l'aplicació o incloure fitxers estàtics (imàtges, CSS, JavaScript, etc).
  Els patrons de URLs s'avaluen en l'ordre que apareixen en aquest fitxer, de dalt a baix. La primera coincidència és la que s'utilitza per a processar la petició.

Existeixen dos tipus de controladors: controladors de seqüencies de comandes i controladors de fitxers estàtics.

5.5.1 Controladors de seqüencies de comandes

Un controlador de seqüencies de comandes executa una seqüencia de comandes en Python com a resposta a una URL en concret. Cada controlador defineix un patró d'URL i la seqüencia de comandes a executar:

- `url`: el patró de la URL, representat amb una expressió regular que pot contenir agrupacions. Per exemple, l'URL `/profile/edit/manager` coincidiria amb l'expresió regular `/profile/(.*?)/(.*)` i utilitzaria `edit` i `manager` com a primera i segona agrupació:

  ```
  handlers:
  - url: /profile/(.*?)/(.*)
    script: /employee/\2/\1.py
  ```

- `script`: la ruta al fitxer a executar, definida des del directori arrel de l'aplicació. Amb l'exemple anterior, el fitxer que s'executaria seria `/employee/manager/edit.py`.

5.5.2 Controladors de fitxers i directoris estàtics

Un controlador de fitxers estàtics retorna el contingut d'un fitxer com a resposta. Els controladors de fitxers estàtics descriuen quins arxius del directori són estàtics i en quines URLs es serveixen.

Aquests controladors permeten servir els fitxers de tal manera que els navegadors els guardin en la seva memòria cau. D'aquesta manera, els navegadors no fan peticions per aquests fitxers, es millora el temps de càrrega de les pàgines i es redueix la transferència de dades. El temps d'expiració d'aquests fitxers es pot definir de forma global per a tots els controladors amb l'element `default_expiration` o bé individulament per a cada controlador amb l'element `expiration`.

Els controladors de fitxers estàtics es defineixen de la següent manera:

```
- url: /static
  static_dir: static
```

En aquest cas, totes les URLs que comencin per `/static` es serviran a partir del directori `static` a l'arrel de l'aplicació. Aquests controladors permeten especificar altres elementes com `mime_type` (el tipus MIME) o `expiration` (el temps d'expiració).

5.5.3 Restricció a usuaris autenticats i administradors

Els controladors disposen d'un paràmetre `login` per a restringir l'accés a usuaris autenticats o usuaris que sigui administradors de l'aplicació. Quan es crida un controlador amb el paràmetre `login`, el controlador comprova que l'usuari estigui autenticat. Si no és així, se'l redirigeix a la pàgina d'accés de Google i un cop s'ha autenticat se'l retorna a la URL de l'aplicació. Si la URL és només per a administradors de l'aplicació i l'usuari no ho és, aquest rebrà un missatge d'error.

A continuació es mostra un exemple de com utilitzar aquest paràmetre:

```
- url: /user
   script: user.py
   login: required

- url: /admin/.*
  script: admin.py
  login: admin
```

En aquest cas, la URL `/user` requereix que l'usuari s'hagi autenticat mentre que les URL `/admin/.*` són només accessibles per a usuaris administradors de l'aplicació.

5.6 Entorn de desenvolupament i SDK

El kit de desenvolupament de software (SDK) de Google App Engine inclou un servidor web que simula l'entorn de Python de google App Engine i permet desenvolupar i provar les aplicacions, i també un script que permet pujar les aplicacions a Google App Engine.

5.6.1 Servidor web de desenvolupament

El servidor web de desenvolupament permet provar les aplicacions en local i reprodueix l'entorn de Google App Engine: restringeix la importació de mòduls, emula la datastore, emula els comptes de Google i en general permet emular tots els seus serveis. L'única cosa que el servidor de desenvolupament no és capaç d'emular són les quotes i límits de l'aplicació, com, per exemple, el temps límit d'una petició.

Un cop instal·lat el SDK de Google App Engine, es pot iniciar el servidor web de desenvolupament amb la comanda `dev_appserver.py`:

```
dev_appserver.py blogyourtravel
```

El servidor web utilitza el port 8080 per defecte i l'aplicació es pot consultar a la URL http://localhost:8080.

Per a canviar el port del servidor web de desenvolupament s'utilitza l'opció `--port`:

```
dev_appserver.py blogyourtravel --port=8000
```

El servidor web de desenvolupament permet l'ús d'altres paràmetres. Per exemple:

```
dev_appserver.py blogyourtravel --port=8000
        --datastore_path=datastore --enable_sendmail
```

En aquest cas, a més a més de canviar el port del servidor, s'activa el `sendmail` per a poder enviar correus electrònics i les dades de la datastore es guarden en un fitxer anomenat `datastore`.

5.6.2 Pujar les aplicacions

El SDK de Google App Engine també inclou un script per a pujar les aplicacions. Per a pujar-les, s'utilitza la comanda `appcfg.py` amb l'acció `update` i el directori de l'aplicació que conté el fitxer `app.yaml`:

```
appcfg.py update blogyourtravel
```

El script `appcfg.py` obté l'identificador de l'aplicació del fitxer app.yaml i demana a l'usuari que s'identifiqui amb el seu correu electrònic i contrasenya del seu compte de Google.

Aquesta comanda també permet altres accions, per exemple:

```
# Elimina els índexos de la datastore
appcfg.py vacuum_indexes blogyourtravel

# Actualitza els índexos de la datastore
appcfg.py update_indexes blogyourtravel

# Descarrega els logs de l'aplicació
appcfg.py request_logs blogyourtravel logs.txt
```

5.7 La interfície d'administració

La interfície d'administració de Google App Engine està disponible a http://appengine.google.com/ o a http://appspot.com.

Aquesta interfície permet:

- Crear noves aplicacions i registar-les amb un domini propi o a appspot.com.
- Veure detalls de les quotes i pagar per a augmentar-les.
- Invitar altres usuaris per a que siguin desenvolupadors de l'aplicació. Aquests usuaris poden accedir a l'administració i pujar noves versions del codi.
- Accedir als logs de l'aplicació i analitzar el tràfic.
- Inspeccionar les dades de la datastore i administrar els índexos.
- Veure i configurar tasques programades.
- Provar noves versions de l'aplicació i canviar la versió activa.

A continuació es mostren dos captures de la interfície d'administració de Google App Engine:

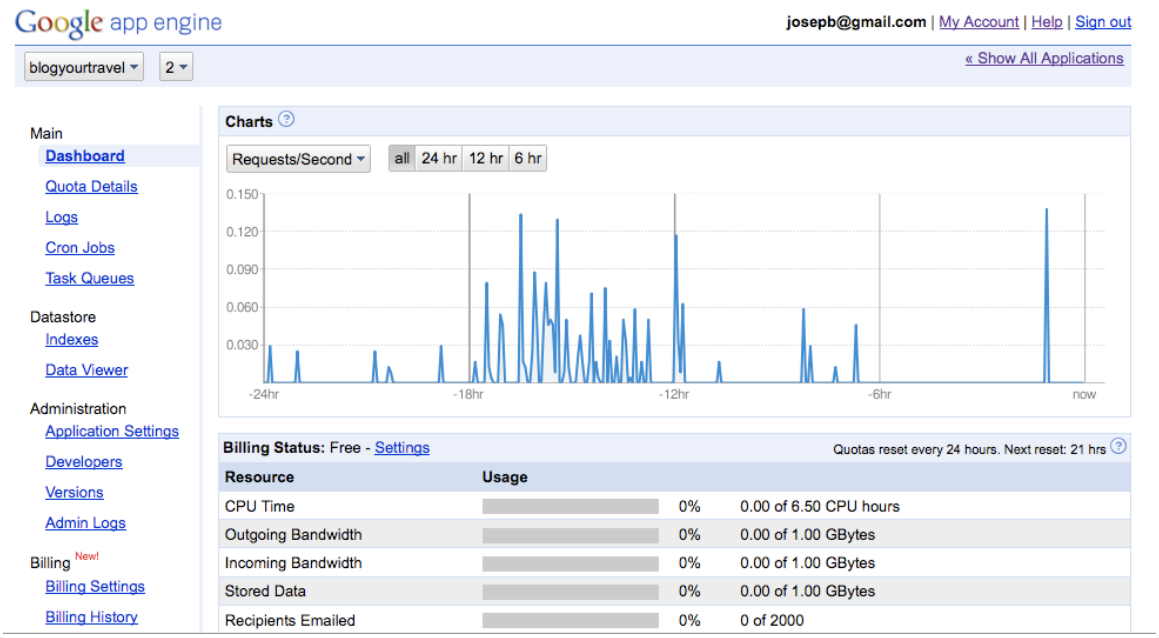

Figura 5.1: *Dashboard* de la interfície d'administració de Google App Engine

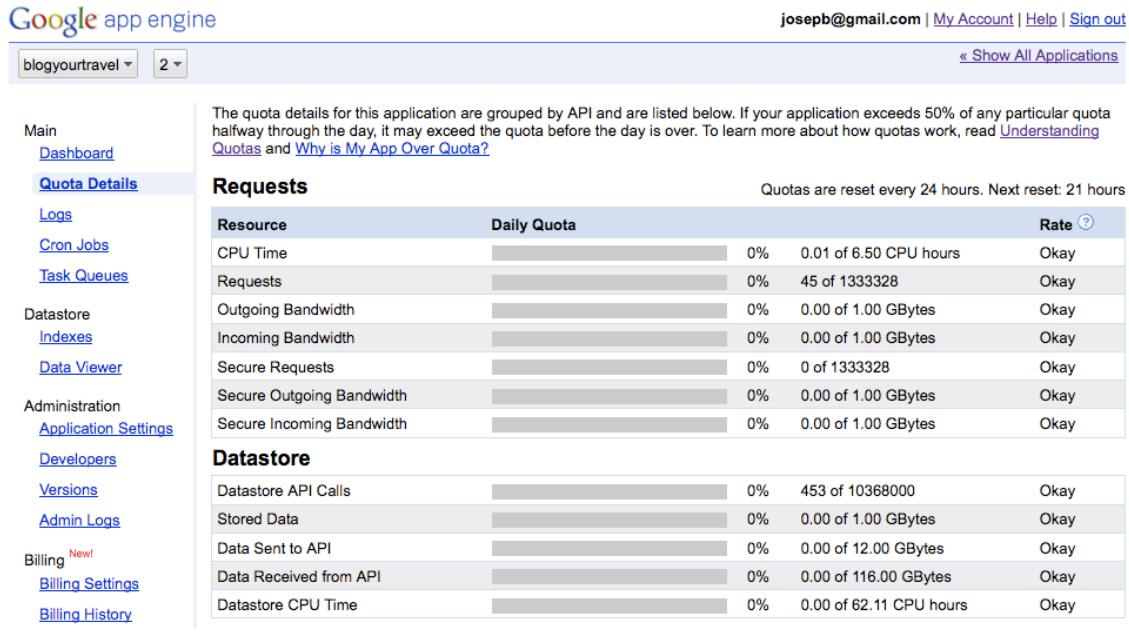

Figura 5.2: Algunes de les quotes de l'aplicació

5.8 Altres eines

5.8.1 El framework webapp

El framework webapp és un framework senzill per a desenvolupar aplicacions web sobre Google App Engine. El framework webapp no és imprescindible per a utilitzar App Engine ja que es permet utilitzar qualsevol aplicació compatible amb WSGI. En aquest treball no s'ha utilitzat aquest framework sinó Django, un framework molt més complet que també és compatible amb Google App Engine. Per aquest motiu, en aquest apartat no s'aprofundeix en aquest framework i només s'expliquen aspectes generals.

El mòdul webapp està inclòs en el paquet `google.appengine.ext`, es facilita juntament amb el SDK i també està disponible en l'entorn de producció. El framework webapp és un mètode senzill per a començar a desenvolupar aplicacions per a Google App Engine i fer proves, però és molt limitat per a aplicacions més grans i complexes.

A continuació es mostra un petit exemple de l'ús del framework webapp:

```
from google.appengine.ext import webapp
from google.appengine.ext.webapp.util import run_wsgi_app

class MainPage(webapp.RequestHandler):
    def get(self):
        self.response.headers['Content-Type'] = 'text/plain'
        self.response.out.write('Hello, webapp World!')

application = webapp.WSGIApplication([('/', MainPage)],
                                     debug=True)
```

```
def main():
    run_wsgi_app(application)

if __name__ == "__main__":
    main()
```

Aquest codi defineix un controlador de peticions (`MainPage`) assignat a la URL arrel (/). Quan webapp rep una petició GET HTTP per a la URL '/', crea una instància de la classe `MainPage` i executa el mètode `get` d'aquest instància. Es pot obtenir informació sobre la petició accedint `self.request` i es poden retornar respostes amb `self.response`.

L'aplicació es representa amb una instància de `webapp.WSGIApplication`. La funció `run_wsgi_app()` utilitza una instància de `webapp.WSGIApplication` (o qualsevol altre objecte d'una aplicació compatible amb WSGI) i l'executa. El funcionament és similar al del mòdul `wsgiref` de la biblioteca estàndard de Python.

Un controlador del framework webapp pot definir diferents mètodes per a processar peticions HTTP: `get()`, `post()`, `put()`, `delete()`, etc.

5.8.2 Logging

Google App Engine utilitza el mòdul `logging` de Python per a fer log dels events que succeixen en l'aplicació. Els logs que genera l'aplicació es poden veure a la interfície d'administració de Google App Engine o en la terminal si s'està fent servir el servidor de desenvolupament.

El mòdul `logging` de Python permet registrar cinc nivells: Debug, Info, Warning, Error i Critical.

A continuació es mostra un exemple de com utilitzar aquest mòdul:

```
import logging
from google.appengine.api import urlfetch

def get_country(point):
    url = 'http://maps.google.com/maps/geo?q=%s&output=csv&hl=en&key=%s'
          % (str(point), settings.GMAPS_KEY)
    try:
        result = urlfetch.fetch(url)
        if result.status_code == 200:
            logging.debug('Status Code 200')
            return result.content.replace('"','').split(',')[-1]
        else:
            logging.debug('Another Status Code')
            return None
    except Exception:
        logging.error('Error in URL Fetch')
        return None
```

5.8.3 App Engine Patch i Google App Engine Django Helper

App Engine Patch i Google App Engine Django Helper són dos projectes de codi lliure que faciliten la creació d'aplicacions de Django sobre Google App Engine.

Django es pot executar a Google App Engine ja que és un framework compatible amb el protocol WSGI i l'App Engine ja inclou una versió de Django, la 0.96. No totes les funcionalitats de Django són compatibles amb Google App Engine. Les diferències més significatives són en el model: la datastore no és una base de dades relacional, que és en el que es basa el ORM de Django. Aquests dos projectes intenten adaptar Django a l'App Engine i permeten fer servir la major part de funcionalitats.

Google App Engine Django Helper va ser el primer projecte en aparèixer, poc després de la publicació de Google App Engine. La solució proposada pel Model es basa en el patró *Adapter* i permet transformar la classe base del Model de Django a la d'App Engine. Això fa automàticament compatibles moltes aplicacions reusables de Django, però també té un gran inconvenient: qualsevol canvi en l'API de la datastore de Google App Engine obliga a una nova versió de Google App Engine Django Helper que incorpori aquests canvis.

App Engine Patch va aparèixer força més tard (febrer del 2009) amb la mateixa idea que Google App Engine Django Helper. La solució que proposa pel Model és simplement fer servir el Model de d'App Engine i adaptar totes aquelles aplicacions de Django que es vulguin portar. És important destacar que aquest projecte adapta l'autenticació de Django als comptes de google, permet utilitzar la seva interfície d'administració en comptes de la de Google App Engine i adapta altres funcionalitats de Django.

Vàries aplicacions de Django ja s'han portat a Google App Engine fent servir aquest projecte i ara per ara sembla millor que Google App Engine Django Helper. Per al desenvolupament de l'aplicació s'ha utilitzat App Engine Patch.

5.8.3.1 App Engine Patch

Amb App Engine Patch, la major part de Django funciona sobre Google App Engine sense modificacions. El canvi més important que cal dur a terme és en el Model, ja que s'ha d'utilitzar la datastore.

Aquest projecte permet que la gran majoria de les funcionalitats de Django es puguin utilitzar a Google App Engine. Per exemple:

- La interfície d'administració de Django.
- L'autenticació de Django, opcionalment integrada amb l'autenticació de Google.
- Les sessions i cache de Django, implementades sobre memcache.
- L'API de formularis, integrada amb el Model de la datastore.
- Les vistes genèriques.

Aquest projecte no adapta únicament Django a Google App Engine sinó que també proporciona un munt d'utilitats que faciliten el desenvolupament. Per exemple:

- Decoradors per a les vistes: `@transaction` i `@login_required`.
- Utilitats per a treballar amb la datastore.

- Integració de l'API remota (`remote_api`) de la datastore amb la comanda de Django `manage.py shell --remote`
- Integració de paquets ZIP.

Capítol 6

Gestió del projecte

6.1 Google Code - Project Hosting

Google Code és un lloc web de Google per a desenvolupadors de programari lliure. Google Code, entre altres, ofereix un servei per a hostatjar projectes de programari lliure semblant a SourceForge.

El servei de hosting de projectes de Google Code funciona sobre la seva infraestructura, permet la revisió de codi, ofereix Subversion i Mercurial, un sistema de gestió d'errors (*issues* o *tickets*), una wiki per a la documentació i descàrregues de fitxers.

Per a utilitzar aquest servei, només és necessari disposar d'un compte de Google i que la llicència del projecte sigui una de les disponibles: Apache License 2.0, Artistic License / GPL, Eclipse License 1.0, GPL v2, GPL v3, LGPL, MIT License, Mozilla Public License 1.1 o New BSD License. La documentació del projecte es pot publicar amb la mateixa llicència o bé amb llicències Creative Commons.

Des del primer moment s'ha fet ús de Google Code per a gestionar tot el projecte. A dia 9 de Juliol de 2009, s'han fet 42 canvis en el codi font i s'han publicat 28 *issues*.

La URL on es pot veure el projecte és http://code.google.com/p/travellog.

Paral·lelament a aquest projecte, s'ha publicat per separat la llibreria per a treballar amb Google Maps des de Python, sota llicència LPGL. Aquest altre projecte està disponible a http://code.google.com/p/googlemapspython

6.2 Documentació

L'aplicació que s'ha desenvolupat en aquest treball no està acabada, és una primera versió que ha servit per a provar la plataforma i les funcionalitats que s'ofereixen són molt bàsiques. Algunes parts del codi s'han d'optimitzar i canviaran força.

En canvi, el codi de la llibreria per a treballar amb Google Maps és molt més estable, es pot utilitzar en qualsevol projecte en Python i pot interessar a més gent.

Per aquest motiu, únicament s'ha documentat la llibreria per a treballar amb Google Maps des de Python. La documentació s'ha fet dins del propi codi Python i a la wiki de Google Code.

6.3 Aspectes legals: llicències

La llicència final de l'aplicació desenvolupada depèn de les llicències del programari utilitzat i la seva compatibilitat. A continuació es mostren les llicències utilitzades:

Programari o llibreria	**Llicència**
Google App Engine SDK	Apache License 2.0
Django	"New" BSD License
App Engine Patch	MIT License
Prototype	MIT License
jQuery	MIT License o GPL
Yahoo Rich Text Editor	"New" BSD License
Dynarch DHTML Calendar	"Free for non comercial use"

Taula 6.1: Llicències del programari utilitzat

La idea inicial era publicar el projecte sota llicència GPLv2. Les llicències MIT i BSD són permisives i són compatibles amb GPL en qualsevol versió. L'única llicència "problemàtica" és l'Apache License 2.0, que és compatible amb GPLv3 però no amb versions anteriors.

Finalment, l'aplicació s'ha publicat sota llicència GPLv3, excepte la classe per a treballar amb Google Maps que s'ha publicat amb llicència LGPLv3.

A banda del programari, en l'aplicació s'utilitzen altres recursos publicats sota llicència Creative Commons:

- Plantilla HTML i CSS, disponible a http://www.infscripts.com
- Pack d'icones "Silk", disponible a http://www.famfamfam.com

Capítol 7

Cas pràctic: Blog Your Travel

En aquest capítol s'expliquen alguns detalls de l'aplicació que s'ha desenvolupat per tal de provar Google App Engine. L'aplicació s'anomena BlogYourTravel i està disponible a http://blogyourtravel.appspot.com.

Nota: quan es va registrar l'aplicació a Google App Engine, el nom de Blog Your Travel estava lliure i no hi havia cap domini registrat amb aquest nom. Dies després va aparèixer BlogYourTravel.com, un servei similar però que no te res a veure amb aquest projecte.

7.1 Anàlisis de requeriments

7.1.1 Requeriments funcionals

A continuació es detallen els requeriments funcionals:

1. **Publicació de viatges**
   L'objectiu principal és que els usuaris registrats puguin publicar els seus viatges en temps real o bé a posteriori.

2. **Mostrar àlbums de fotografies**
   Els viatges s'han de poder enllaçar amb àlbums de fotografies online de Picasa.

3. **Mostrar mapes i rutes**
   Els viatges han de tenir mapes amb punts ubicats i rutes marcades.

7.1.2 Requeriments no funcionals

1. **Escalabilitat**
   L'aplicació ha de ser fàcilment escalable ja que està oberta a múltiples usuaris.

2. **Límits de l'App Engine**
   Google App Engine té uns límits que l'aplicació ha de complir: emmagatzematge de dades, temps de processament, nombre de peticions...

3. **Integració**
   L'aplicació s'ha d'integrar amb altres APIs de Google com Picasa o Maps.

4. **Compatibilitat**
   L'aplicació web ha de ser compatible amb els principals navegadors del mercat.

5. **Codi lliure**
   Tan el codi de l'aplicació com la documentació del projecte s'han de publicar sota una llicència lliure.

7.2 Casos d'ús

L'objectiu principal d'aquest treball és estudiar la plataforma i no pas dissenyar aquesta aplicació. És per això que en aquesta secció només es descriuen alguns dels casos d'ús de l'aplicació.

7.2.1 Actors

- **Usuari registrat:** s'han registrat utilitzant el seu compte de google i poden afegir continguts.
- **Visitant:** poden consultar els blogs i viatges publicats.

7.2.2 Descripció dels casos d'ús

- **Cas d'ús: Registrar blog**.
  - **Descripció:** permet als visitants registrar el seu blog.
  - **Actors:** visitant.
  - **Precondicions:** el visitant s'ha autenticat a l'aplicació utilitzant el seu compte de google. L'usuari no té cap blog creat.
  - **Seqüència normal:**
    1. El visitant accedeix a la url */register*
    2. El visitant omple el formulari de registre.
    3. Es comprova si existeix algun blog amb la url introduïda o aquesta és invàlida.
    4. S'emmagatzema el blog a la base de dades.
  - **Seqüencia alternativa:**
    3. Si ja existeix algun blog amb la url introduïda o és invàlida es demana a l'usuari que la canvïi.
  - **Postcondicions:** El blog s'afegeix a la base de dades.
- **Cas d'ús: Crear nou viatge**.
  - **Descripció:** permet als usuaris registrats crear un nou viatge.
  - **Actors:** usuari registrat.
  - **Precondicions:** l'usuari s'ha autenticat a l'aplicació utilitzant el seu compte de google i té un blog creat.
  - **Seqüència normal:**
    1. L'usuari accedeix a l'enllaç per afegir un nou viatge.
    2. L'usuari omple les dades del viatge i envia el formulari.
    3. Es comprova si les dades són correctes.

4. S'emmagatzema el viatge a la base de dades.

– **Seqüencia alternativa:**

3. Si alguna dada és incorrecta es demana a l'usuari que ho modifiqui.

– **Postcondicions:** El viatge s'afegeix a la base de dades.

- **Cas d'ús: Crear nova entrada**.

– **Descripció:** permet als usuaris registrats crear una nova entrada d'un viatge.

– **Actors:** usuari registrat.

– **Precondicions:** l'usuari s'ha autenticat a l'aplicació utilitzant el seu compte de google, té un blog registrat i com a mínim un viatge creat.

– **Seqüència normal:**

1. L'usuari accedeix a l'enllaç per afegir una nova entrada.
2. L'usuari omple les dades de l'entrada.
3. A partir de la localització que ha escrit l'usuari s'obtenen les coordenades i es mostra el punt en el mapa.
4. L'usuari envia el formulari.
5. Es comprova si les dades són correctes.
6. S'emmagatzema l'entrada a la base de dades.

– **Seqüencia alternativa:**

3. L'usuari també pot introduir directament les coordenades del punt.
5. Si alguna dada és incorrecta es demana a l'usuari que ho modifiqui.

– **Postcondicions:** L'entrada s'afegeix a la base de dades.

- **Cas d'ús: Afegir fotografies**.

– **Descripció:** permet als usuaris registrats afegir fotografies d'una entrada de viatge.

– **Actors:** usuari registrat.

– **Precondicions:** l'usuari s'ha autenticat a l'aplicació utilitzant el seu compte de google, té un blog creat i com a mínim una entrada de viatge.

– **Seqüència normal:**

1. L'usuari accedeix a l'enllaç per afegir fotografies d'una entrada.
2. L'usuari sel·lecciona un dels seus àlbum de picasa.
3. L'usuari té l'opció de filtrar les fotografies per tag o data.
4. El sistema obté les fotografies a partir de les dades que ha introduït l'usuari.
5. L'usuari sel·lecciona quines de les fotografies vol importar.
6. El sistema importa les fotografies (URLs de la fotografia i miniatura) a la base de dades.

– **Seqüencia alternativa:**

4. Si no es troba cap fotografia se li comunica a l'usuari.

– **Postcondicions:** Les fotografies s'han afegit a la base de dades.

7.3 Model de domini

Aquesta aplicació s'ha realitzat seguint un desenvolupament iteratiu. Inicialment es va desenvolupar la base de l'aplicació i a partir d'aquí s'han anat fent iteracions, afegint noves funcionalitats i corregint errors.

Aquest és el diagrama de classes de l'aplicació en el seu inici:

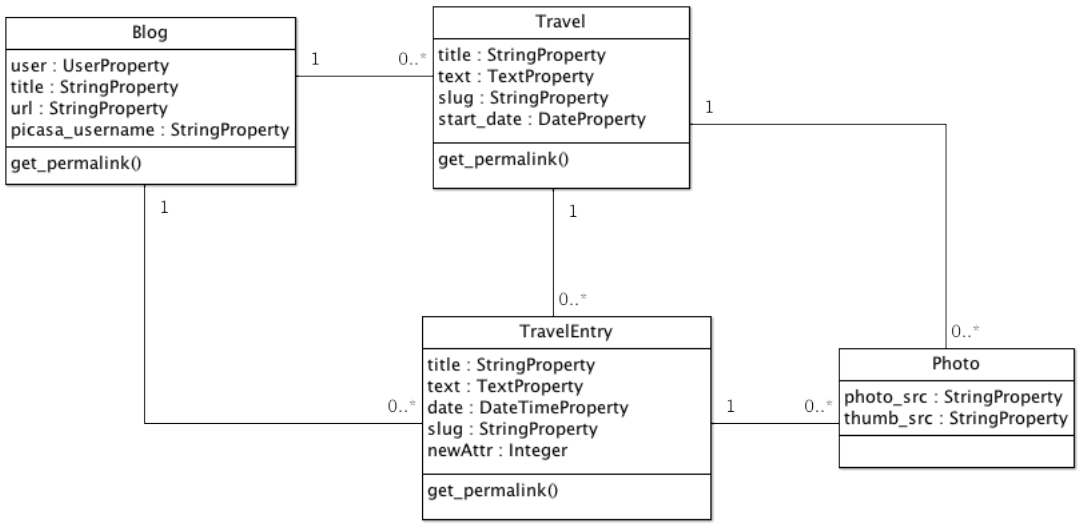

Figura 7.1: Diagrama de classes inicial

Aquest diagrama mostra només els mètodes i atributs principals de les classes. Cal tenir en compte que aquestes classes hereten de la classe `db.Model` de Google App Engine i hereten tots els mètodes necessaris per treballar amb la datastore.

El model de domini de l'aplicació és senzill i les relacions són fàcils d'entendre:

- L'aplicació permet registrar blogs. Cada blog pertany a un únic usuari (de Google) i un usuari només pot tenir un blog.
- Cada blog pot tenir varis viatges
- Cada viatge pot tenir vàries entrades.
- Cada entrada pot tenir vàries fotografies. Són fotografies de Picasa o Flickr i es guarden les URLs.
- Les relacions entre `Travel-Photo` i `Blog-TravelEntry` s'han introduït per a reduir els accessos a la datastore.

A partir d'aquest disseny inicial, l'aplicació ha anat evolucionant i el diagrama de classes final és el següent:

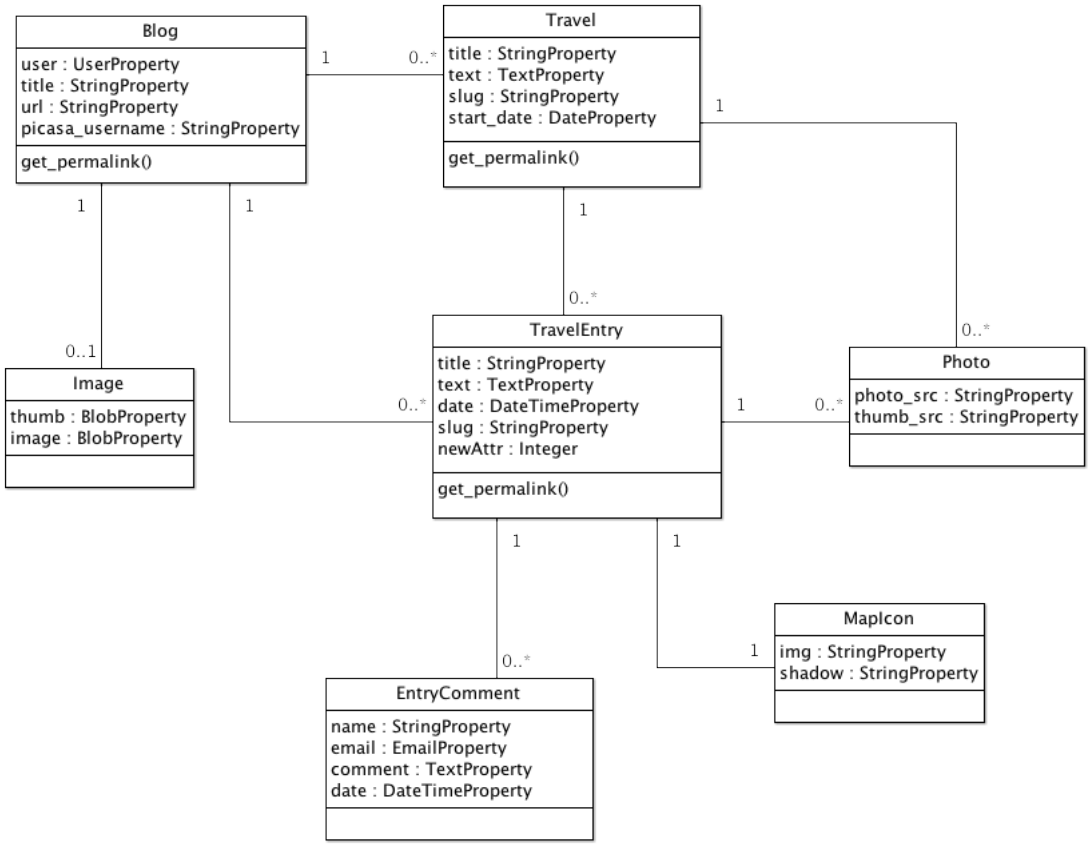

Figura 7.2: Diagrama de classes final

Les relacions que apareixen noves són les següents:

- Cada blog pot tenir una imatge associada. Aquesta imatge la puja l'usuari i es guarda a la datastore.
- Cada entrada permet escollir la icona que es mostrarà en el mapa.
- Cada entrada pot tenir varis comentaris dels visitants del blog.

7.4 Patrons de disseny

Els patrons de disseny busquen solucions a problemes comuns en el desenvolupament de software. Un patró de disseny és una solució a un determinat problema de disseny.

En l'aplicació desenvolupada s'utilitzen principalment dos patrons de disseny: el patró Model Vista Controlador com a patró arquitectural i el patró Active Record per a l'accés a base de dades.

7.4.1 Model Vista Controlador (MVC)

Model Vista Controlador (MVC) és un patró d'arquitectura de software que separa les dades d'una aplicació, la interfície d'usuari i la lògica de control en tres components. D'aquesta manera, es poden fer canvis a un d'aquests components amb un impacte mínim sobre la resta.

Aquest patró divideix l'aplicació en tres components:

- **Model:** és la representació de la informació i les dades sobre les quals funciona l'aplicació. Sovint, aquesta capa també s'anomena domini. En aquest treball, la capa Model la proporciona Google App Engine amb la datastore.
- **Vista:** la capa vista correspon a la presentació que es mostra a l'usuari i amb la qual interactua. Aquesta vista pot ser una pàgina web, una aplicació amb una GUI o fins i tot vistes específiques per a terminals mòbils. En aquest treball, la capa Vista és XHTML i la proporciona Django amb el seu sistema de plantilles.
- **Controlador:** és la capa que respon als events i s'encarrega de fer canvis en el model i la vista. En aquest treball, la capa Controlador la proporciona Django.

El patró MVC és àmpliament utilitzat en aplicacions web. Django es basa en aquest patró tot i que l'anomenen MTV (Model Template View).

7.4.2 Active Record

Active Record és un patró de disseny que s'utilitza per a l'accés a la base de dades. Aquest patró implementa el mapeig objecte-relacional (ORM) i permet persistir objectes en una base de dades relacional. Habitualment, cada taula de la base de dades correspon a una classe i cada fila de la taula a un objecte d'aquesta classe.

Generalment, cada classe que implementi aquest patró haurà de proporcionar com a mínim mètodes per a:

- Crear noves instàncies.
- Actualitzar instàncies
- Esborrar instàncies.
- Realitzar consultes.

El patró Active Record s'ha fet popular a partir de Ruby on Rails, tot i que frameworks com Django també l'implementen. La datastore de Google App Engine també utilitza aquest patró.

7.4.3 Decorator

El patró decorador permet modificar el comportament d'un objecte en temps d'execució. Quan s'aplica el factor decorador, l'objecte original no sap que està sent decorat i rep una nova funcionalitat.

Els decoradors en Python ofereixen una forma elegant d'alterar el comportament d'una classe, mètodes, i propietats. Un decorador en Python és una funció que rep una funció com a paràmetre i retorna una nova funció com a resultat.

En l'aplicació, un dels decoradors que s'utilitza és per comprovar si l'usuari està autenticat en les vistes de la part d'aministració. Aquest decorador és de Django i està adaptat a l'autenticació de Google App Engine:

```
from django.contrib.auth.decorators import login_required
from google.appengine.api import users

@login_required
def register(request):
    user = users.get_current_user()
    ...
```

7.5 Aspectes de la implementació

7.5.1 Llibreria per a Google Maps

Per a facilitar la integració amb Google Maps s'ha creat una classe en Python que permet mostrar mapes sense necessitat d'escriure codi Javascript. D'aquesta manera, es crea el mapa en codi Python i es genera el codi Javascript corresponent.

A continuació es mostra un exemple del seu ús:

```
from django.conf import settings
from travellog.models import *
from travellog import maps, util

def get_travel_map(travel):
    map = maps.Map(settings.GMAPS_KEY, 'map')
    for entry in travel.entries:
        if entry.point:
            icon = maps.Icon(entry.icon.img, entry.icon.shadow)
            entry_text = ["<h2>%s</h2>" % entry.title]
            entry_text.append("<h4>%s</h4>" % entry.date)
            entry_text.append("<br/><a href=\"%s\">Go to this entry</a>"
                              % entry.get_permalink())
            lat, lon = util.get_coordinates(entry.point)
            map.addPoint(maps.Point(lat, lon, ''.join(entry_text), icon))
    return map.printMap()
```

El codi de l'exemple anterior genera el mapa d'un viatge, amb totes les seves entrades marcades en el mapa.

Aquesta llibreria s'ha publicat a Google Code com a un nou projecte a part, ja que es pot utilitzar en altres aplicacions en Python. Aquest projecte està publicat sota llicència LGPLv3 i està disponible a la següent URL: http://code.google.com/p/googlemapspython

7.5.2 Google Data APIs

GData és l'acrònim de Google Data APIs, un protocol estàndard per a comunicar-se amb els serveis de Google. Entre d'altres, GData permet comunicar-se amb: Google Analytics, Google Apps, Google Calendar, Google Docs, Picasa Web Albums o Youtube. Aquest protocol es basa en altres protocols com REST, ATOM o RSS. Hi ha implementacions de GData en varis llenguatges de programació: Python, PHP, Java, .NET, etc.

En aquest cas, s'ha utilitzat la implementació de GData en Python per a integrar Picasa Web Albums amb l'aplicació. A continuació es mostra un exemple de l'ús de l'API:

```
import atom
import gdata.urlfetch
import gdata
import gdata.photos.service
import gdata.media
import gdata.geo
import gdata.alt.appengine

def get_albums(username):
    gd_client = gdata.photos.service.PhotosService()
    gdata.alt.appengine.run_on_appengine(gd_client)
    albums = gd_client.GetUserFeed(user = str(username))
    return [(album.gphoto_id.text, album.title.text)
            for album in albums.entry]
```

En aquest exemple, la funció `get_albums()` obté els àlbums públics de l'usuari que es rep com a paràmetre.

7.5.3 Internacionalització

L'aplicació està disponible en tres idiomes: català, castellà i anglès. Per a internacionalitzar l'aplicació s'ha utilitzat la internacionalització de Django.

Si es vol traduir l'aplicació a un nou idioma n'hi ha prou en crear un nou fitxer de missatges per aquest idioma, traduir les cadenes i compilar-ho. Per exemple, si es vol afegir l'alemà:

```
django-admin.py makemessages -l de
django-admin.py compilemessages
```

7.5.4 Gestió i autenticació d'usuaris

L'autenticació d'usuaris que s'ha utilitzat és la que proporciona Google App Engine amb els comptes de Google. Els avantatges d'utilitzar aquest sistema són molts tan pels usuaris com pels desenvolupadors i ja s'han comentat anteriorment.

Google App Engine Patch permet integrar l'autenticació d'usuaris de Google amb l'autenticació bàsica de Django.

7.5.5 Ús dels serveis de Google App Engine

Tal com s'ha dit en la introducció, un dels objectius d'aquest treball és desenvolupar una aplicació sobre Google App Engine que permeti estudiar la plataforma. Per aquest motiu, s'ha intentat utilitzar el màxim de serveis que proporciona la plataforma:

- **Datastore:** s'utilitza en tota l'aplicació per a emmagatzemar dades.
- **Comptes de Google:** s'utilitzen per a l'autenticació d'usuaris. Cada blog pertany a un únic compte de Google.
- **Memcache:** s'utilitza a la pàgina principal de l'aplicació, per a mostar els últims blogs registrats i entrades publicades sense tenir que accedir a la datastore.
- **Email:** s'envia un email als usuaris quan es registren amb la URL del seu blog.
- **Url Fetch:** s'utilitza en la funció `get_country`, que obté un país a partir d'unes coordenades.
- **Imatges:** l'API d'imatges s'utilitza per a mostrar una imatge de l'usuari al lateral del blog. L'usuari pot afegir aquesta imatge a l'apartat "About your blog"a l'administració. Aquesta API també s'hauria pogut utilitzar en les entrades del blog, per tal de permetre als usuaris pujar una fotografia amb cada entrada. Aquesta opció no s'ha contemplat per a no sobrepassar els límits d'espai de la datastore.

7.6 URLs de l'aplicació

Una de les característiques principals de l'aplicació i que la diferencia de la resta és l'organització de les URLs. Les URLs estan pensades per a que siguin fàcils de recordar pels usuaris i a la vegada donen informació del contingut que es mostra en cada pàgina. Aquest tipus d'URLs són possibles gràcies a Django.

A continuació es mostren varis exemples de URLs de l'aplicació:

- http://blogyourtravel.com/register: mostra la pàgina per a registrar un blog.
- http://blogyourtravel.com/admin: permet accedir a l'administració del blog, afegir viatges, entrades, fotografies, etc.
- http://blogyourtravel.com/polecito: pàgina principal del blog de l'usuari 'polecito'. Es mostren les seves últimes entrades, una fotografia de l'usuari i el mapa del seu últim viatge.
- http://blogyourtravel.com/polecito/travels: es mostren els viatges de l'usuari.
- http://blogyourtravel.com/polecito/norge07: es mostren les entrades d'un dels viatges de l'usuari.
- http://blogyourtravel.appspot.com/polecito/norge07/oslo-vigekand-park: es mostra una entrada en concret de l'usuari. Si l'entrada té fotografies, es mostren en una galeria feta amb Javascript. Igualment, si l'entrada té comentaris, també es mostren.

- http://blogyourtravel.appspot.com/polecito/about: es mostra una pàgina sobre el blog de l'usuari.
- http://blogyourtravel.appspot.com/polecito/map: es mostra un mapa global de l'usuari amb totes les seves entrades.

Aquestes URLs es configuren al fitxer `urls.py` emprant expresions regulars. A continuació es mostren algunes de les URLs d'aquest fitxer:

```
from django.conf.urls.defaults import *
from django.views.generic.simple import direct_to_template

urlpatterns = patterns('',
    (r'^$', direct_to_template, {'template': 'main.html'}),
    (r'^register/$', 'travellog.views.register'),
    (r'^admin/$','travellog.admin.blog_welcome'),
    (r'^admin/blog/about/$','travellog.admin.blog_about'),
    (r'^admin/blog/settings/$','travellog.admin.blog_settings'),
    (r'^admin/blog/delete/$','travellog.admin.blog_delete'),
    (r'^admin/entries/$','travellog.admin.entry_list'),
    (r'^admin/entries/new/$','travellog.admin.entry_add'),
    (r'^admin/entries/delete/$', 'travellog.admin.entry_delete'),
    (r'^admin/travels/$','travellog.admin.travel_list'),
    (r'^admin/travels/new/$','travellog.admin.travel_add'),
    (r'^admin/travels/delete/$', 'travellog.admin.travel_delete'),
    (r'^admin/travels/(?P<travel_key>[^\.^/]+)/$',
        'travellog.admin.travel_detail'),
    (r'^(?P<blog_key>[^\.^/]+)/$', 'travellog.views.blog_main'),
    (r'^(?P<blog_key>[^\.^/]+)/map/$', 'travellog.views.global_map'),
    (r'^(?P<blog_key>[^\.^/]+)/about/$', 'travellog.views.blog_about'),
    (r'^(?P<blog_key>[^\.^/]+)/travels/$', 'travellog.views.travel_list'),
    (r'^(?P<blog_key>[^\.^/]+)/(?P<travel_key>[^\.^/]+)/$',
        'travellog.views.travel_detail'),
    (r'^(?P<blog_key>[^\.^/]+)/(?P<travel_key>[^\.^/]+)/
        (?P<entry_key>[^\.^/]+)', 'travellog.views.entry_detail')
)
```

7.7 Errors coneguts

Des que es va publicar la primera versió de l'aplicació, s'han anat corregint errors i aspectes que no funcionaven correctament. No obstant, hi ha alguns errors que s'han identificat però que encara no s'han pogut solventar:

- A l'hora d'importar fotografies de Picasa Web Albums, si l'àlbum té moltes fotografies es pot produir un *timeout*. Això és degut al temps límit d'una petició que imposa Google App Engine i que ara mateix no es pot modificar.
- Quan un usuari introdueix una entrada no es té en compte la seva zona horària.
- L'aplicació en general està poc optimitzada i alguna vegada, en algun accés a la datastore, es sobrepassa el temps límit i es produeix un error.

De totes formes, sempre que es produeix un error, Django captura l'excepció que s'ha llençat i es mostra una pàgina d'error personalitzada. A més a més, els errors queden guardats en els logs de l'aplicació.

7.8 Captures de l'aplicació

En aquesta secció es mostren algunes captures de pantalla de l'aplicació:

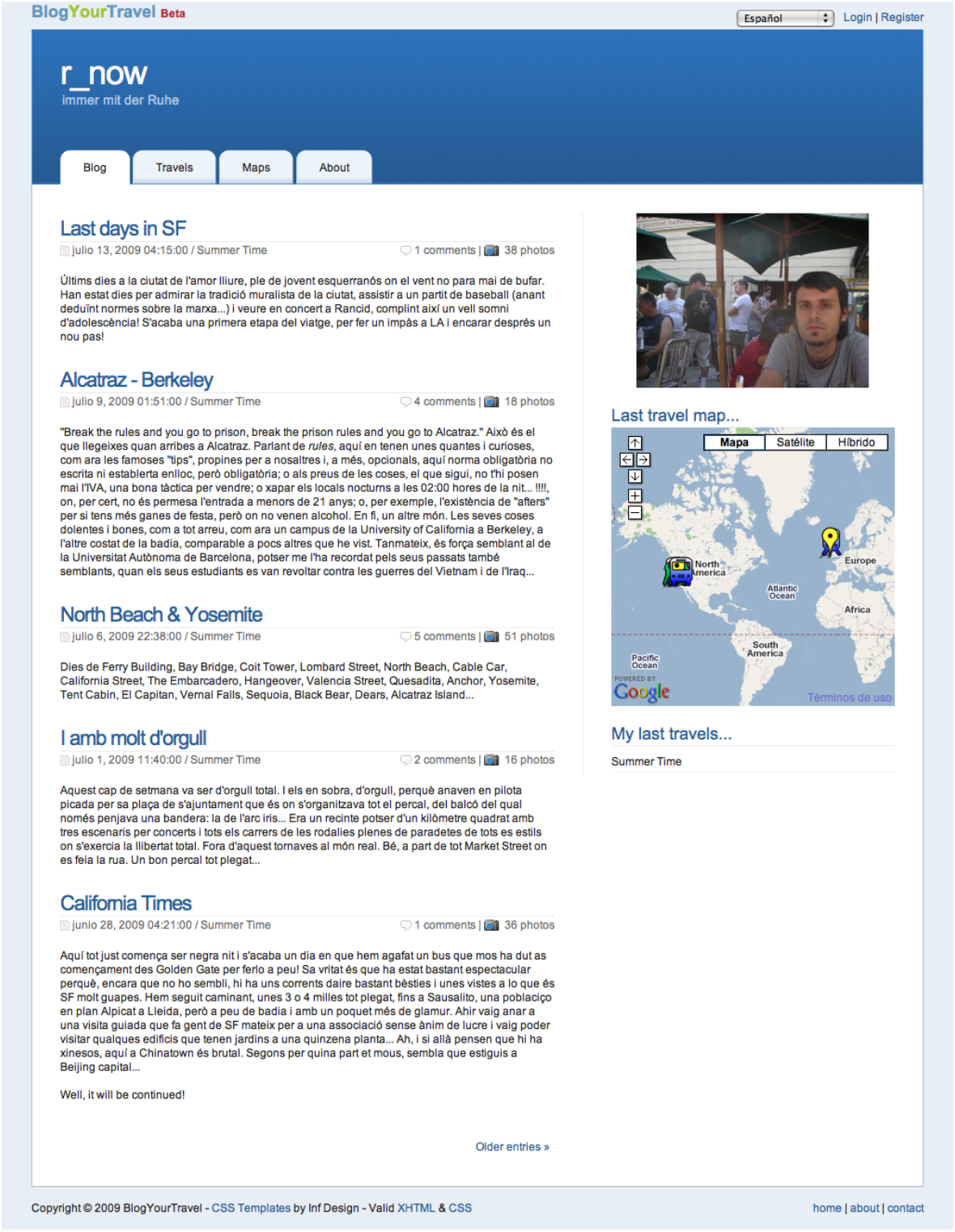

Figura 7.3: Pàgina inicial del blog d'un usuari

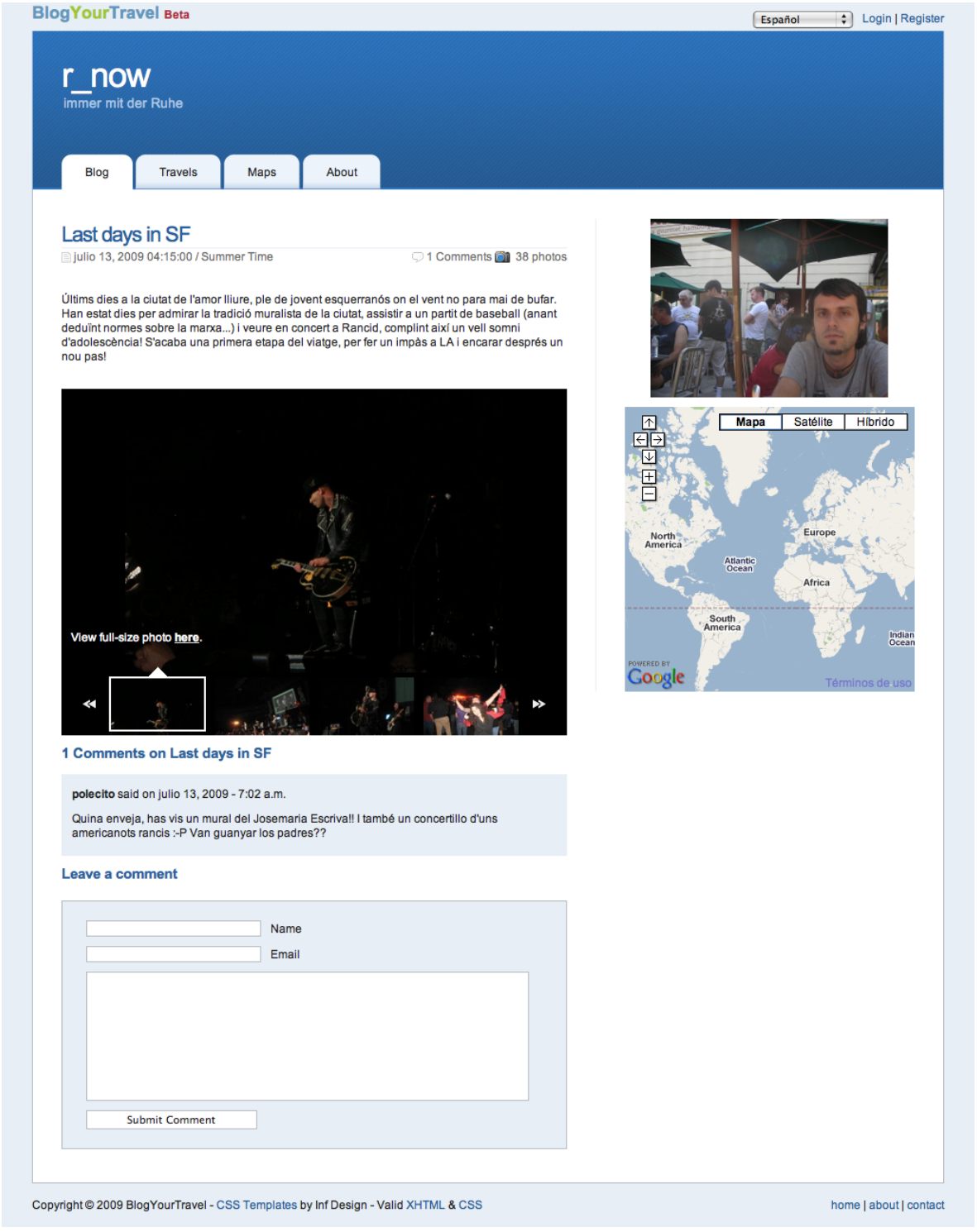

Figura 7.4: Detall d'una entrada d'un usuari

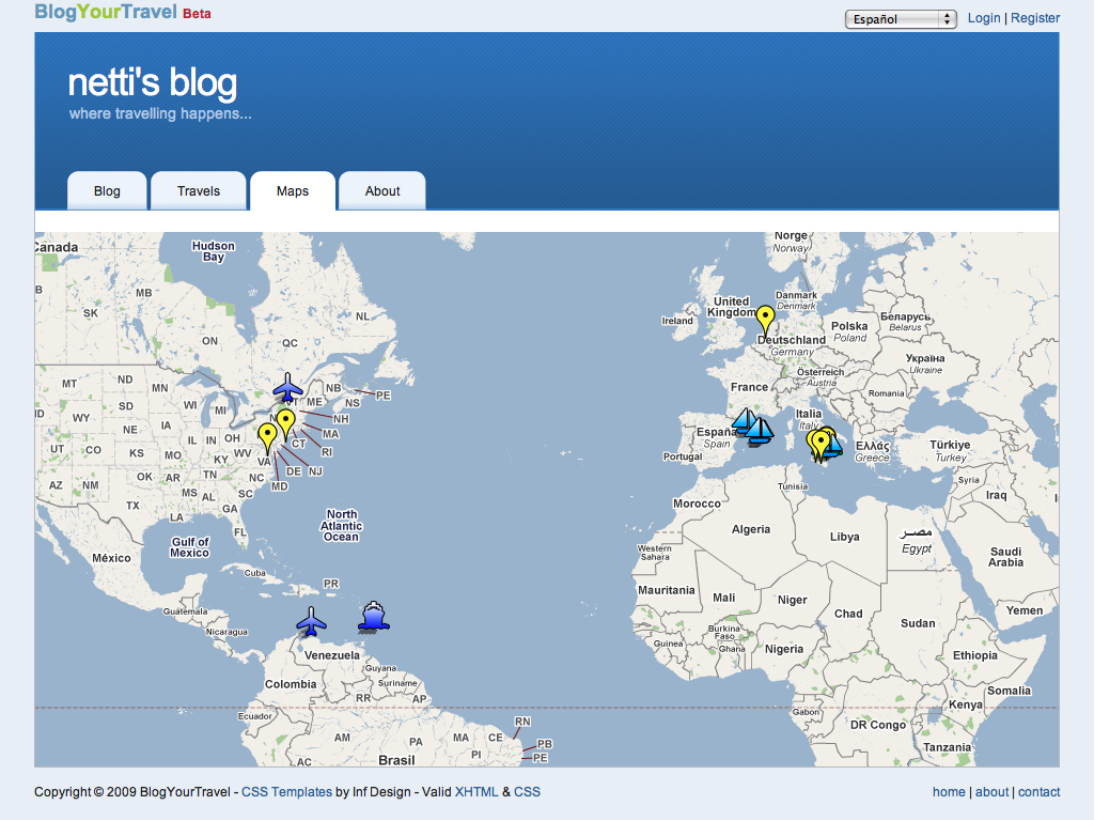

Figura 7.5: Mapa amb totes les entrades de l'usuari

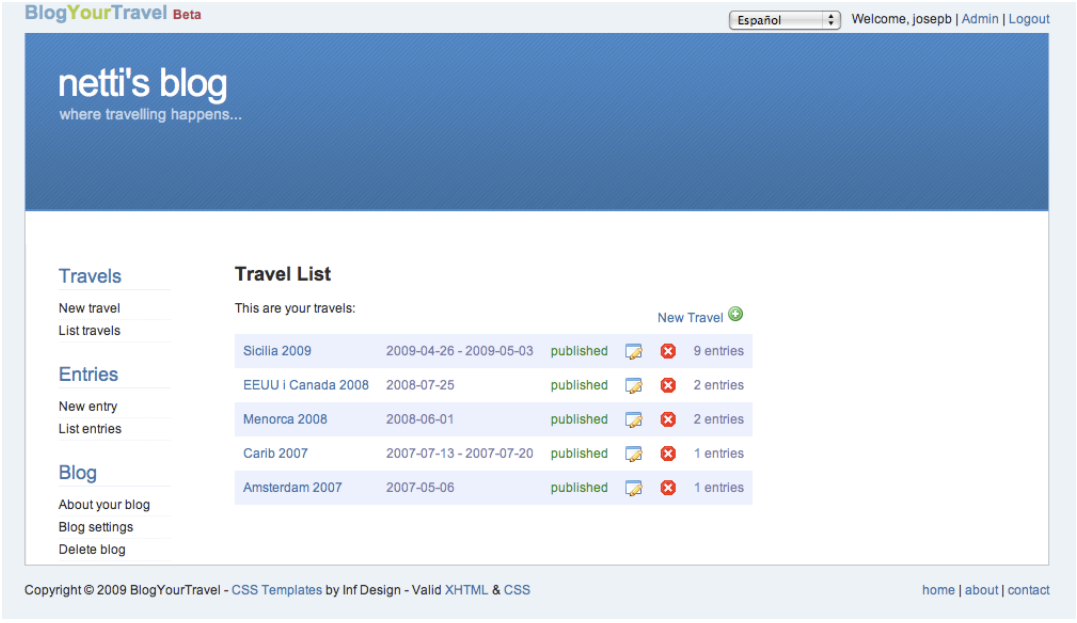

Figura 7.6: Administració d'un blog

Capítol 8

Conclusions i treball futur

8.1 Conclusions

Experiència i valoració personal

A nivell personal, aquest treball ha estat una bona experiència. He tingut l'oportunitat d'estudiar tecnologies actuals, aplicar els coneixements adquirits en aquests dos anys de màster i fins i tot aplicar coneixements més transversals.

Un dels meus objectius personals era que aquest treball fos una petita síntesi del que he estudiat en aquest màster. En aquest sentit, puc dir que ho he complert: en aquest treball s'hi veuen reflectits aspectes de Llenguatges Interpretats, d'Introducció al Programari Lliure, de Comunitats i Projectes, d'Aspectes Legals, d'Enginyeria del Software, de Desenvolupament d'Aplicacions i Serveis Web i d'altres matèries en menor mesura.

A més a més, en les assignatures de Plataformes de Desenvlupament d'Aplicacions de Comerç Electrònic i d'Anàlisi de la Web, he tingut la sort de poder relacionar les pràctiques amb aquest treball. Això ha suposat una motivació extra ja que era una tema que m'interessava. Fruit d'això n'és, per exemple, l'Annex on es comparen les tecnologies utilitzades en aquest treball amb una possible equivalència en Java.

Després d'haver provat i estudiat aquestes tecnologies, tinc clar que les utilitzaré d'ara en endavant: Google App Engine és una bona opció per a qualsevol projecte personal i Django és un framework molt potent que intentaré utilitzar en projectes professionals.

En aquest treball també he pogut aprofundir en moltes eines útils en el desenvolupament web i integrar-les: jQuery, Prototype, Yahoo! UI Library, Google Maps, Google Data APIs, etc. També he pogut millorar els meus coneixements de llenguatges com Python, Javascript, HTML o CSS, tot i que aquests dos últims continuen sense ser el meu fort.

De ben segur que l'experiència d'aquest treball i tots aquests coneixements adquirits em seran útils en el futur.

Els problemes del cloud computing

El cloud computing és un dels fenòmens de moda i moltes empreses ofereixen serveis basats en aquest tipus de computació. A l'hora d'endinsar-se en aquest món i escollir un d'aquests serveis cal tenir en compte els seus inconvenients.

Per una banda, els avantatges del cloud computing són clars: les aplicacions s'executen en una infraestructura distribuïda, escalen amb més facilitat, no cal preoucupar-se del manteniment dels servidors, es poden integrar fàcilment amb altres serveis i tenen una alta disponibilitat.

D'altra banda, els principal problema del cloud computing també es veu ràpidament. Només és necessari un tall del servei durant un període de temps per adonar-se de la dependència total del proveïdor. Aquesta dependència no és només perl la disponibilitat del servei, sinó també per les barres de sortida que hi ha. La falta d'estàndards i mecanismes a l'hora de fer una migració no ajuda, i marxar d'un proveïdor pot ser molt dificil o fins i tot impossible.

Si es decideix apostar per una plataforma de cloud computing cal tenir en compte aspectes de seguretat, control, privacitat... i sobretot, és important basar el desenvolupament en eines que siguin compatibles en altres entorns i intentar evitar els serveis exclusius de la plataforma.

Google App Engine

Avui en dia, Google App Engine és una opció perfecta per a una *start-up* o per a un projecte personal, situacions en les que es vol fer una prova sense gastar recursos. Si l'aplicació té èxit, es poden comprar més recursos i l'aplicació pot creixer sense dificultats.

Si és una empresa la que vol desenvolupar les seves aplicacions sobre Google App Engine, hi ha molts més dubtes. Per una banda, els preus que ofereix Google App Engine són molt competitius i alguns dels seus serveis són molt útils; però per l'altra banda, la plataforma té moltes mancances per a desenvolupar aplicacions comercials. Hi ha certes limitacions en les aplicacions, com l'allotjament de fitxers o el temps límit de les peticions.

Sovint les empreses necessiten un control directe de les seves aplicacions i alguna persona que respongui quan el servei no funciona. Ara mateix, Google App Engine no ofereix cap garantia de funcionament o de que les dades no es perdin, i moltes empreses no s'ho poden permetre. No obstant, el temps de disponibilitat (*uptime*) que pot oferir Google és molt superior al de qualsevol allotjament tradicional.

Cal tenir en compte que Google App Engine és una plataforma molt nova i que fins fa poc estava en fase de proves. Poc a poc s'han anat corregint errors, escoltant les opinions dels desenvolupadors i afegint millores. S'espera que es continuï així i la plataforma evolucioni.

La potència de Django

En aquesta aplicació queda demostrada la potència de Django. Django permet un desenvolupament ràpid, àgil, senzill i elegant. No només proporciona una arquitectura

Model Vista Controlador (tot i que en Django s'anomena Model Template View) sinó que també soluciona algunes de les tasques més habituals en el desenvolupament web: l'administració automàtica, l'API de formularis, les vistes genèriques... De Django es diu que és un framework *Batteries Included* (amb piles incloses) per la gran quantitat de llibreries extra que inclou.

Avui en dia, Django és un dels frameworks de desenvolupament web més potents i complets que existeixen.

L'aposta de Google pel programari lliure

El programari lliure té un paper molt important a Google i la pròpia empresa hi dedica molts recursos. Entre d'altres, cal destacar els següent projectes:

- Google Chrome: navegador web de codi lliure.
- Android: plataforma lliure per a dispositius mòbils.
- Google Code: aplicació per a hostatjar el codi.

A part d'aquests, també és important destacar Google Summer of Code, una iniciativa de Google per a que els estudiants puguin fer pràctiques durant l'estiu treballant en algun projecte de programari lliure.

Contrariament al que es pot pensar, la tecnologia més important de Google i sobre la que es desenvolupen tots els seus serveis, es propietaria. Google File System i BigTable són la base de la infraestructura distribuïda i escalable de Google, i aquesta és una de les característiques que els diferencia dels seus competidors.

La importància del programari lliure en el món web

L'aplicació que s'ha desenvolupat en aquest treball és una mostra de la importància del programari lliure en el desenvolupament web. El programari lliure no s'utilitza únicament en la infraestructura (servidors, dns, firewall, correu...) sinó que també hi ha disponibles moltíssimes eines per al desenvolupament web.

El programari lliure s'extén per tot el món web i per tots els llenguatges de programació: gestors de continguts (Drupal, Plone, TYPO3, OpenCMS...), frameworks (Django, Ruby on Rails, Symfony, Spring...), llibreries per a JavaScript i Ajax (jQuery, Prototype, Mootools, Dojo...) o eines de publicació de blogs, on Wordpress és el dominant.

Aquest treball és només una petita mostra de la gran quantitat d'eines de programari lliure de qualitat que existeixen. Avui en dia, la major part d'aplicacions i portals web estan desenvolupats utilitzant alguna eina lliure.

Les llicències de programari lliure

En aquest treball s'integren altres projectes de programari lliure ja existents: el SDK de Google App Engine, Django, Google App Engine Patch, jQuery, Prototype... A l'hora de combinar el codi d'aquests projectes apareixen dubtes respecte la compatibilitat de llicències i en quins casos s'apliquen.

Tot i que des de fa uns anys s'ha aturat la proliferació de llicències lliures, n'hi ha massa i algunes d'elles són pràcticament igual. A més, algunes clàusules estan pensades per aplicar-se en llenguatges compilats, on el programari es distribueix en un únic paquet, i no pas en llenguatges interpretats com Python.

La diversitat de llicències lliures és un dels grans impediments per a que el programari lliure s'extengui encara més.

8.2 Treball futur i possibles millores

En aquest treball només s'empren algunes de les possibilitats de Google App Engine. L'aplicació desenvolupada ha servit "d'excusa" per a poder estudiar aquesta plataforma, Django i altres tecnologies. Si es vol que l'aplicació pugui ser una alternativa a d'altres semblants, cal continuar-hi treballant i fer-hi millores.

A continuació es mostren només algunes de les possibles millores per a l'aplicació. La tecnologia utilitzada és molt nova, dinàmica i encara té molt recorregut, i les possibilitats de l'aplicació són molt grans i es pot integrar amb molts serveis. Només és una qüestió de temps, imaginació i idees.

8.2.1 Millores en l'aplicació per a publicar blogs

Les funcionalitats de l'aplicació són les més bàsiques i imprescindibles per a poder publicar un blog de viatges. Respecte a l'aplicació, el treball futur és enorme i s'han d'afegir moltes millores: funcionalitats pròpies dels blogs i funcionalitats relacionades amb els viatges. Per exemple:

- Publicar les entrades d'un blog en RSS i ATOM per a que fos més fàcil seguir-lo.
- Poder etiquetar els continguts (viatges, entrades, fotos) amb *tags*.
- Moderar els comentaris de les entrades i rebre'ls per email.
- L'usuari hauria de poder escollir entre vàries plantilles per al seu blog i personalitzar-ne els colors, tipus de lletra, imatges, etc.
- L'aplicació hauria de poder importar fotografies d'àlbums de Flickr ja que és un servei molt utilitzat.
- Els mapes s'haurien de poder personalitzar: escollir el nivell de zoom, el tipus de mapa, dibuixar-hi línies...
- Les fotografies importades es podrien mostrar ubicades en el mapa.
- L'aplicació també s'hauria d'integrar amb serveis com Youtube o Google Earth.
- Afegir un nou tipus de contingut: rutes amb punts des de GPS.
- Integrar l'aplicació amb altres serveis populars com Twitter o Facebook.

8.2.2 Millores en la tecnologia

Les millores en la tecnologia utilizada són tan o més importants que afegir noves funcionalitats a l'aplicació. Aquestes van des d'optimitzar l'aplicació a utilitzar noves APIs publicades:

Optimització de l'aplicació

Primer de tot, és molt important optimitzar l'aplicació per intentar no sobrepassar les quotes gratuïtes. És necessari analitzar els punts on l'aplicació gasta més recursos i reduir-los. Cal tenir en compte les característiques d'una base de dades d'objectes i aprofitar-les:

- La datastore és lenta a l'hora de serialitzar i deserialitzar els objectes. Com més petits són els objectes, més ràpides són les consultes. Dividir les classes del Model en classes més petites milloraria el temps d'accés a la base de dades.
- On la datastore és més ràpida és en les consultes a partir de la clau d'un objecte. Si la consulta es fa filtrant per altres camps, la datastore està obligada a obtenir moltes més entitats i comparar-les una per una. Si s'aconsegueix que les consultes a la datastore es facin a partir de la clau, el nombre d'accessos a la base de dades serà inferior i el temps d'accés també millorarà.
- També es pot utilitzar l'API de Memcache en les pàgines de l'aplicació més consultades per a reduir els accessos a la datastore.

Integració amb GeoDjango

GeoDjango és una branca de Django que proporciona un framework per a desenvolupar aplicacions web utilitzant Sistemes d'Informació Geogràfica. Seria interessant estudiar aquest projecte i mirar d'integrar-ho amb l'aplicació.

Noves versions del programari

Avui en dia la tecnologia evoluciona molt ràpidament i sovint apareixen noves versions del programari. Començant per Google App Engine, en els últims mesos s'ha publicat una API de tasques, una API per accedir remotament a la datastore, un nou entorn d'execució en Java o la possibilitat de realitzar tasques programades. En aquesta treball, aquestes funcionalitats no s'han estudiat. A més a més, en el *roadmap* de Google App Engine hi ha un nou servei per emmagatzemar fitxers i això seria molt útil per a l'aplicació.

Igualment, és important estar atent a la publicació de noves versions de Django, Google App Engine Patch i altres projectes similars que ajuden a portar les aplicacions de Django a Google App Engine.

De la mateixa manera, Google Maps i Picasa Web Albums són serveis de Google que es milloren continuament i algunes d'aquestes noves funcionalitats es poden aplicar a l'aplicació. Per exemple, fa poc s'ha publicat la versió 3 de Google Maps que podria utilitzar en aquest treball.

Annex: Comparació amb tecnologia Java

Introducció

En aquest annex es compara la tecnologia utilitzada en aquest treball (principalment Google App Engine i Django) amb una possible equivalència en Java, amb Spring i JPA. Es comparen les tres capes del patró Model Vista Controlador i algunes llibreries comunes.

La informació que s'inclou prové d'un treball realitzat per a l'assignatura Plataformes de Comerç Electrònic. Alguns exemples de codi en Python són de l'aplicació web desenvolupada en el Treball Final de Màster i de la documentació de Django. De la mateixa manera, alguns dels exemples de codi en Java estan aprofitats d'una pràctica desenvolupada a l'assignatura Desenvolupament d'Aplicacions i Serveis Web. En tots dos casos, el codi s'ha reduït i simplificat per a que fos més fàcil entendre'l.

Model

La capa Model és segurament la que té més diferències entre les dues tecnologies. En bona part això es degut a que la base de dades que proporciona Google App Engine és orientada a objectes, mentre que els frameworks JPA o Hibernate estan preparats per treballar amb bases de dades relacionals.

Java - Spring

JPA (Java Persistence API) es la API estàndard de persistència d'objectes de Java. L'objectiu d'aquesta API és mantenir els avantatges de la orientació a objectes i poder interactuar amb bases de dades relacionals. Per això proporciona una eina de mapeig objecte-relacional. JPA converteix les dades entre els tipus utilitzats per Java i els definits per SQL i també s'encarrega de generar sentències SQL per a tractar aquestes dades.

Els objectes que es vulguin persistir han de ser POJOs (Plain Old Java Object), classes simples que no depenen d'un framework en especial. El mapeig objecte-relacional es pot definir amb anotancions dins de les pròpies classes o bé amb fitxers descriptors XML.

Exemple:

```
@Entity
@Table(name="travel")
public class Travel {

  @Id
  @GeneratedValue(strategy = GenerationType.SEQUENCE)
  protected int id_travel;
  protected String title;
  @OneToMany(cascade=ALL, mappedBy="travel")
  protected List<Entry> entries;

  public String getTitle(){
    return this.title;
  }

  public void setTitle(String title){
    this.title = title;
  }

  public List<Entry> getEntries() {
    return this.entries;
  }

  public void setEntries(List<Entry> entries){
    this.entries = entries;
  }

}


@Entity
@Table(name="entry")
public class Entry {

  @Id
  @GeneratedValue(strategy = GenerationType.SEQUENCE)
  protected int id_entry;
  protected String title;
  protected String text;

  @ManyToOne
  @JoinColumn(name="id_travel")
  protected Travel travel;

  public String getTitle(){
    return this.title;
  }

  public void setTitle(String title){
    this.title = title;
  }

  public String getText(){
    return this.text;
  }

  public void setText(String text){
```

```
    this.text = text;
  }

  public Travel getTravel(){
    return this.travel;
  }

  public void setTrabel(Travel travel){
    this.travel = travel;
  }

}
```

Per a mapejar les classes i els seus atributs s'utiltiza el patró Convention over Configuration, que té com a objectiu simplificar el desenvolupament. En aquest cas, el paradigma Convention over Configuration de la següent manera: si no s'especifica el contrari, la taula corresponent a la classe Travel s'anomenarà igual, i el mateix passarà amb els seus atributs i els seus tipus.

Tradicionalment, la major part de frameworks de persistència, com Hibernate, necessitaven un fitxer de configuració XML per a cada classe que es volia persistir. En l'actualitat, molts frameworks utilitzen aquest paradigma: JPA, Ruby on Rails, Django, Zend Framework, CakePHP, symfony...

El més habitual per a obtenir les dades del model és utilitzar el patró DAO. Tot i que aquest patró inicialment sorgeix per la necessitat de gestionar diverses fonts de dades, també s'utilitza per encapsular la forma d'obtenir aquestes dades.

Per a aplicar el patró DAO amb Spring cal heretar de la classe JpaDaoSupport:

```
public class TravelJPADAO extends JpaDaoSupport {

  public void addTravel(Travel travel) {
    getJpaTemplate().persist(travel);
  }
  public void deleteTravel(int id_travel) {
    getJpaTemplate().remove(getTravel(id_travel));
  }
  public void updateTravel(Travel travel) {
    getJpaTemplate().merge(travel);
  }

  public List<Travel> getTravels() {
    return getJpaTemplate().find("select t from Travel t");
  }
  public Travel getTravel(int id_travel) {
    return getJpaTemplate().find(Travel.class, travel);
  }

}
```

App Engine - Django

La Datastore és l'eina que proporciona Google App Engine per a persistir objectes. No és una base de dades relacional, és orientada a objectes i utilitza una arquitectura distribuïda per a escalar grans quantitats de dades.

Les classes que s'han de persistir a la Datastore s'anomenen entitats. Una entitat té una o més propietats de diferents tipus de dades: enters, decimals, cadenes de text, dates, etc. Aquestes classes han d'heretar de la classe db.Model i tots els seus atributs han de ser instàncies d'alguna subclasse de Property.

Exemple:

```
from google.appengine.ext import db

class Travel(db.Model):
  title = db.StringProperty(required = True)
  start_date = db.DateProperty()
  end_date = db.DateProperty()
  published = db.BooleanProperty(default = False)

class Entry(db.Model):
  travel = db.ReferenceProperty(Travel,collection_name='entries')
  title = db.StringProperty(required = True)
  text = db.StringProperty()
```

Per a obtenir dades del model no s'utilitza el patró DAO sinó el patró Active Record. Els objectes, en heretar de la classe db.Model, hereten tots els mètodes necessaris per a obtenir, crear, modificar o esborrar noves instàncies:

```
#Crear una instància de la classe Travel
travel = Travel(title = 'Viatge a Sicilia', published = True)

#Crear una instància de la classe Entry
entry = Entry(title = 'Palermo', travel = travel)

#Guardar les instàncies
travel.put()
entry.put()

#Obtenir una instància a partir de la seva clau
key = entry.key()
entity = db.get(key)

#Esborrar una instància
entity.delete()

#Obtenir totes les instàncies de la classe Entry
entries = Entry.all()
```

El mètode 'put()' serveix per a emmagatzemar un nou objecte a la Datastore o actualitzar-ne un de ja existent. La Datastore se n'encarrega internament.

El patró Active Record seria dificil d'adoptar en Java perquè és incompatible amb el concepte de POJO. Aquesta idea es basa en que les classes del domini no han de dependre d'altres classes del framework i, per tant, en són independents. En aques cas, si es fes servir el patró Active Record, les classes del domini a persistir haurien d'heretar d'una classe del framework. Com que en Java no existeix l'herència múltiple, aquestes classes ja no podrien heretar de cap més.

Controlador

Tant App Engine - Django com Spring utilitzen el patró Front Controller: les URLs es mapegen a una determinada classe (o mètodes d'aquella classe) que actuen com a controladors. Aquestes classes implementen la lògica de l'aplicació i obtenen les dades que es motraran a les vistes.

Java - Spring

El controlador de Spring és la classe DispatcherServlet que es mapeja al fixer web.xml i on s'especifica quin patró seguiran les URLs:

```
<servlet>
  <servlet-name>example</servlet-name>
  <servlet-class>
    org.springframework.web.servlet.DispatcherServlet
  </servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
  <servlet-name>example</servlet-name>
  <url-pattern>*.htm</url-pattern>
</servlet-mapping>
```

En l'exemple anterior, totes les peticions que acabin amb .htm les resoldrà el DispatcherServlet example. El fitxer example-servlet.xml contindrà, entre d'altres, totes les url de l'aplicació mapejades a un controlador en concret:

```
<bean name="/index.htm" class="example.controller.HomePageController"></bean>
```

Cada controlador correspon a una classe Java que implementa alguna de les interfícies de controladors de Spring:

```
public class HomePageController implements Controller{
  public ModelAndView handleRequest(HttpServletRequest arg0,
    HttpServletResponse arg1) throws Exception {
    return new ModelAndView("index.jsp");
  }
}
```

A part de Controller, hi ha d'altres interfícies per a diferents propòsits del controlador. Per exemple:

- SimpleFormController: per a mostrar, validar i tractar formularis.
- AbstractWizardFormController: per a crear wizards (formularis amb varis passos).

Aquests controladors s'encarreguen d'obtenir totes les dades necessàries i retornen un objecte ModelAndView que correspon a una vista amb les dades en el context.

App Engine - Django

En el cas d'una aplicació amb App Engine i Django la capa Controlador s'anomena Vista però el funcionament és similar.

A Google App Engine, l'equivalència al fitxer de configuració web.xml d'una aplicació web Java és el fitxer app.yaml. En aquest fitxer, a banda de la configuració de l'aplicació, es defineixen les URLs i els scripts que les han de tractar:

```
handlers:

- url: /.*
  script: main.py
```

En aquest fitxer main.py és on es crea una aplicació de Django i actua com a FrontController. A l'igual que Spring, Django carrega el fitxer urls.py on es mapegen totes les URLs de l'aplicació a una vista (controlador) en concret:

```
from django.conf.urls.defaults import *

urlpatterns = patterns('',
  (r'^hello/$', 'views.hello'))
```

La vista que correspon a aquesta url és la funció hello del fitxer views.py:

```
from django.http import HttpResponse

def hello(request):
  return HttpResponse("Hello world")
```

La configuració de URLs de Django permet crear URLs dinàmiques de forma ràpida, senzilla i elegant:

```
urlpatterns = patterns('',
  (r'^example/(?P<number>\d+)/$', views.example),
)
```

En aquest cas, totes les URLs que concordin amb l'expressió regular es mapejaran a la vista example del fitxer views.py. La funció example rebrà un paràmetre extra que correspon a la cadena capturada entre parèntesis al patró especificat:

```
def example(request, number):
  return HttpResponse("You entered %s" % number)
```

Si la URL fos /example/2 la variable number valdria '2'. En canvi, si pensem en altres llenguatges com PHP o Java, el més comú seria tenir una url com example.htm?number=2 o /example?number=2.

El gestor de URLs de Django permet que la nostra aplicació tingui el que s'anomenen *clean* o *friendly* URLs mentre que amb altres llenguatges caldria utilitzar el mòdul RewriteRule d'Apache o similar per aconseguir el mateix resultat. Pel que fa als controladors, els de Django són més tolerants a canvis ja que corresponen a simples funcions mentre que en Spring cada controlador ha d'implementar una interfície en concret segons el seu pròpoist.

Vista

La capa Vista és probablement la que té més semblances entre les dues tecnologies. En el cas de Django la capa vista s'anomena Template.

En tots dos casos el sistema de plantilles permet combinar codi html i codi dinàmic. Els sistemes de plantilles defineixen una plantilla base i diferents blocs. La resta de plantilles extenen aquesta plantilla base i poden sobreescriure aquests blocs.

Java - Spring

El sistema de plantilles de Struts Tiles es defineix en un fitxer XML. Aquest fitxer conté totes les vistes de l'aplicació. En el següent exemple, es defineix una plantilla base amb quatre blocs que la resta de plantilles podran sobreescriure:

```
<tiles-definitions>
  <definition name="template" template="/templates/template.jsp">
    <put-attribute name="header" value="/templates/header.jsp" />
    <put-attribute name="left" value="/templates/left.jsp" />
    <put-attribute name="content" value="/templates/default.jsp" />
    <put-attribute name="footer" value="/templates/footer.jsp" />
  </definition>
  <definition name="index" extends="template">
    <put-attribute name="content" value="/templates/index.jsp" />
  </definition>
</tiles-definitions>
```

Tal com s'ha especificat al fitxer de configuració, la plantilla base (template.jsp) ha de definir els quatre blocs indicats:

```
<%@ taglib uri="http://tiles.apache.org/tags-tiles" prefix="tiles" %>
<html>
  <head>
    <title>Example</title>
  </head>
  <body>
    <div id="main_container">
      <div id="header">
        <tiles:insertAttribute name="header" />
      </div>
```

```
      <div id="left">
        <tiles:insertAttribute name="left" />
      </div>
      <div id="content">
        <tiles:insertAttribute name="content" />
      </div>
      <div id="footer">
        <tiles:insertAttribute name="footer" />
      </div>
    </div>
  </body>
</html>
```

Quan una vista retorna la plantilla ïndex", el sistema de plantilles Tiles carrega la plantilla base "template.jspï en el bloc çontent"hi carrega la plantilla ïndex.jsp".

Les plantilles en JSP poden contenir codi java i etiquetes.

App Engine - Django

En Django, la idea del sistema de plantilles és similar a l'anterior. En aquest cas, no existeix un fitxer de configuració on definir totes les pàgina sinó que és en la pròpia pàgina on s'especifica de quina plantilla s'hereta i quins blocs es sobreescriuen.

L'estructura que seguiran les plantilles es defineix en una plantilla base com la següent (base.html):

```
<html>
  <head>
    <title>Example</title>
  </head>
  <body>
    <div id="main_container">
      <div id="header">
        {% block header %}
          Header text
        {% endblock %}
      </div>
      <div id="left">
        {% block left %}
          Left text
        {% endblock %}
      </div>
      <div id="content">
        {% block content %}
        {% endblock %}
      </div>
      <div id="footer">
        {% block footer %}
          Footer text
        {% endblock %}
      </div>
    </div>
  </body>
</html>
```

La plantilla que correspon a la pàgina principal ïndex.html"és la següent:

```
{% extends 'base.html' %}

{% block content %}
  <h1>Example</h1>
{% endblock %}
```

Aquesta plantilla sobreescriu el block çontentï la resta de blocs s'heren de la plantilla "base.html".

Les plantilles de Django poden contenir etiquetes (codi pseudo-Python) per tal de mostrar continguts dinàmics:

```
{% for object in object_list %}
  {{ object.title }}
{% endfor %}
```

Altres

La majoria de frameworks de desenvolupament d'aplicacions web proporcionen llibreries o mòduls que permeten realitzar tot tipus de tasques. En el cas de Django, es diu que és un framework "Batteries Included"(amb piles incloses), fent referència a les moltes llibreries i funcionalitats extra que aporta. En aquesta secció es comparen algunes d'aquestes llibreries.

Interfície d'administració

Una de les parts més potents de Django és la seva interfície d'administració automàtica que permet realitzar les operacions tradicionals CRUD (Create, Read, Update, Delete).

A partir de les propietats de les classes del Model i de metadades, Django crea una interfície que es pot utilitzar perfectament en entorns de producció. L'administració permet definir l'ordre dels registres, filtres, camps de cerca, registres relacionats...

Google App Engine també proporciona una interfície d'administració automàtica però molt més limitada que la de Django. En el cas de Spring no hi ha cap eina per administrar les classes del Model.

Formularis

Java - Spring

Per a crear un formulari en Spring cal editar varis fitxers i crear-ne de nous ja que el codi per a processar el formulari es divideix en vàries parts.

Al fitxer de configuració on es defineixen tots els controladors, s'afegeix un nou controlador per a la pàgina amb un formulari:

```
<bean name="/admin/addHotel.htm"
    class="cat.udl.apsweb.controller.AdminNewHotelController">
  <property name="commandName" value="hotel"/>
  <property name="commandClass" value="cat.udl.apsweb.bean.Hotel"/>
    <property name="formView" value="addHotel.jsp"/>
  <property name="successView" value="redirect:/admin/index.htm" />
  <property name="validator">
    <bean class="cat.udl.apsweb.validator.HotelValidator" />
  </property>
</bean>
```

Al controlador se li injecten vàries propietats:

- commandName: el nom de la variable que contindrà les dades del formulari.
- commandClass: la classe d'aquesta que representa aquestes dades.
- formView: la vista encarregada de mostrar el formulari.
- successView: la vista que es mostrarà un cop enviat el formulari.
- validator: la classe encarregada de validar el formulari.

El funcionament del formulari és el següent:

- Quan es faci una petició a '/admin/addHotel.htm', aquesta serà processada pel controlador cat.udl.apsweb.controller.AdminNewHotelController.
- Aquest controlador crearà una instància de la classe cat.udl.apsweb.bean.Hotel que representarà les dades del formulari.
- El controlador mostarà la pàgina 'addHotel.jsp' on es mostarà el formulari buit per a introduir-hi les dades.
- Quan s'enviï el formulari, Spring farà una crida a la classe cat.udl.apsweb.validator.HotelValidator que s'encarregarà de validar les dades.
- Si les dades són incorrectes, es mostrarà de nou la pàgina 'addHotel.jsp' amb els errors que s'han detectat.
- Si les dades són correctes, es cridarà al mètode onSubmit del controlador de la pàgina que s'encarregarà de processar el formulari.
- Finalment, es redirigirà a la pàgina '/admin/index.htm'.

La classe corresponent al controlador, en aquest cas, hereta de la classe SimpleFormController i redefineix el mètode onSubmit que és el que s'executa quan s'envia el formulari:

```
public class AdminNewHotelController extends SimpleFormController {

  public AdminNewHotelController() {
    setCommandClass(Hotel.class);
    setCommandName("hotel");
```

```
  }

  protected ModelAndView onSubmit(Object command) throws Exception {
    Hotel hotel = (Hotel) command;
    /* Processar el formulari */
    return new ModelAndView(getSuccessView());
  }

}
```

La classe que s'encarregarà de validar les dades és la següent:

```
public class HotelValidator implements Validator {

  public boolean supports(Class arg0) {
    return Hotel.class.equals(arg0);
  }

  public void validate(Object obj, Errors errors) {
    Hotel hotel = (Hotel) obj;
    ValidationUtils.rejectIfEmptyOrWhitespace(errors, "name",
       "required.name", "El camp nom és obligatori.");
    ValidationUtils.rejectIfEmptyOrWhitespace(errors, "city",
      "required.city", "El camp ciutat és obligatori");
    ValidationUtils.rejectIfEmptyOrWhitespace(errors, "address",
       "required.address", "El camp adreça és obligatori");
    ValidationUtils.rejectIfEmptyOrWhitespace(errors, "description",
      "required.description", "El camp descripció és obligatori");

  }

}
```

El mètode supports serveix per a indicar quins objectes és capaç de validar aquesta classe. En aquest cas, els objectes que valida la classe HotelValidator són de la classe Hotel. L'altre mètode, validate, és el que es cridarà a l'hora de validar el formulari. Aquest mètode comprova que cap camp del formulari estigui buit i especifica els missatges d'error que es mostraran per a cada camp.

Finalment, la plantilla que s'encarregarà de mostrar el formulari és la següent:

```
<form:form method="POST" action="addHotel.htm" commandName="hotel">
  <label>Nom: </label>
  <form:input path="name" />
  <form:errors path="name"  cssClass="error"/>

  <label>Ciutat: </label>
  <form:input path="city" />
    <form:errors path="city"  cssClass="error"/>

  <label>Adreça: </label>
  <form:input path="address" />
  <form:errors path="address"  cssClass="error"/>

  <label>Descripció:  </label>
  <form:textarea path="description" />
```

```
  <form:errors path="description"  cssClass="error"/>

    <input type="submit" value="Afegir"/>

</form:form>
```

En la plantilla s'utilitzen els tags 'form' de Spring que permeten fer el `binding` de cada camp a l'atribut que correspon de la classe Hotel. El tag 'form:errors' és el que s'encarrega de mostrar els missatges d'error d'un determinat camp.

App Engine - Django

8.2.2.0.1 Per a crear i validar un formulari en Spring ha estat necessari editar o crear 4 fitxers diferents, i sobretot, hem hagut de crear el formulari pràcticament de zero quan exactament corresponia a una classe del Model (la classe Hotel).

En el cas de Django, la API de formulari és també una de les més potents, però a més a més, és molt senzilla i pràctica. Els formularis hereten de la classe django.forms.Form i la forma en que es declaren els camps és molt semblant a les classes del Model:

```
from django import forms

class ContactForm(forms.Form):
  name = forms.CharField(max\_length=100, initial='Your name')
  message = forms.CharField(widget=forms.Textarea)
  sender = forms.EmailField(help\_text='Enter a valid email address')
  cc_myself = forms.BooleanField(required=False, label='Copy for myself?')
```

Un formulari es composa per un o més camps que corresponen a instàncies d'una de les subclasses de la classe `django.forms.Field`. En aquest cas, CharField, EmailField i BooleanField són els únics tipus de camps que utilitzem. Cada tipus de camp s'encarrega de validar les dades entrants, mostrar un widget adeqüat, convertir les dades a tipus de Python i validar-les. Els arguments que es passen a cada camp permeten especificar les seves característiques i, de cara a la validació, dir si el camp és obligatori o no.

Per a processar el formulari s'utilitza una vista de Django, que correspon a una funció Python:

```
def contact(request):
  if request.method == 'POST': # El formulari s'ha enviat
    form = ContactForm(request.POST)
        if form.is_valid(): # El formulari és vàlid
            # Processar el formulari
            return HttpResponseRedirect('/thanks/')
    else:
        form = ContactForm() # Es crea el formulari buit.

    return render_to_response('contact.html', {
        'form': form,
    })
```

La validació d'un formulari es realitza quan es crida al mètode `is_valid`. Per defecte es valida que les dades enviades coincideixin amb els tipus esperats i que els camps obligatoris hi siguin presents. Si es vol afegir altres regles de validació es pot fer definint el mètode clean de la classe del formulari o amb el mètode clean_<nom_del_camp>() per a validar un camp en concret:

```
class ContactForm(forms.Form):
  ...
  def clean_email(self):
    invalid_emails = ['test@example.com','test@test.com']
    if self.cleaned_data['email'] in invalid_email:
      raise forms.ValidationError('This email is not valid.')
    return self.cleaned_data['email']
```

Una de les característiques de Django és que permet crear formularis a partir de classes del Model. Per a fer-ho, la classe del formulari ha d'heretar de la classe django.forms.ModelForm, definir la classe del Model i els camps que es volen excloure o incloure:

```
class CommentForm(ModelForm):
  class Meta:
    model = Comment
    fields = ('name','comment')
```

Per a mostrar els formularis a les plantilles n'hi ha prou en cridar al mètode `as_p` del formulari.

```
<form action="/contact" method="POST">
  {{ form.as_p }}
  <input type="submit" value="Submit" />
</form>
```

L'etiqueta `form.as_p` mostra cada camp del formulari en un paràgraf. Els formularis també es poden representar amb `form.as_ul` (per a representar-ho amb una llista), `form.as_table` (per a representar-ho amb una taula) o bé mostrar els camps individualment.

Conclusions

Avui en dia, el més important a l'hora de desenvolupar una aplicació web no és el llenguatge de programació escollit sinó les eines i frameworks disponibles en aquest llenguatge. Tant Django com Spring són dos frameworks molt potents que faciliten el desenvolupament d'aplicacions web però amb filosofies diferents:

- Django és un framework de programació web en Python que busca un desenvolupament ràpid, senzill i basat en el principi DRY (Don't Repeat Yourself).
- Spring és un framework de programació web que facilita el desenvolupament en J2EE. És molt extensible i permet integrar-hi altres solucions ja existents.

La principal diferència és la orientació de Django al desenvolupament ràpid i àgil enfront a l'extensibilitat i mantenibiltiat de Spring.

Django és àgil ja que permet que el desenvolupador es concentri en la lògica de l'aplicació i s'oblidi de tasques més habituals. Django automatitza tasques repetitives en la majoria de projectes:

- Interfície d'administració automàtica a partir del Model.
- Sistema d'autenticació basat en usuaris, grups i permisos.
- Construcció i validació de formularis.

En canvi, Spring no proporciona cap de les funcionalitats anteriors. A més a més, les aplicacions J2EE en general no són tan àgils:

- Hi ha molts fitxers de configuració en XML.
- És necessari compilar tot el codi i generar un .war.
- Sovint, és necessari reiniciar el servidor d'aplicacions Tomcat.

A favor de les aplicacions en J2EE i Spring, cal destacar que hi ha molta documentació i que la llibreria de Java és molt extensa. També proporcionen IDEs que faciliten i "agilitzen"el desenvolupament:

- Autocompletat de codi: getters, setters, etc.
- Facilitat en compilar i desplegar l'aplicació.
- Permeten debugar fàcilment.
- Existeixen plugins, wizards...

No obstant, sovint els desenvolupadors que programen en Java es veuen lligats a un determinat IDE pels motius anteriors. En canvi, en Django es pot desenvolupar amb un simple editor de text, però és més dificil trobar un bon editor per a Python que faciliti el desenvolupament.

Resumint la comparació de les diferents capes del Model Vista Controlador per separat:

- **Model:** tots dos frameworks proporcionen un ORM, fant servir el patró Convention over Configuration i eviten l'ús de SQL. La única diferència clara és que Django proporciona una interfície d'administració automàtica a partir del Model mentre que en Spring caldria integrar alguna altra eina o fer-ho manualment.
- **Vista:** és segurament la capa que té més similituds entre els dos frameworks. Tots dos sistemes es basen en l'herència de plantilles i permeten definir tags propis. Malgrat que Django permet l'ús d'un llenguatge pseudo-Python per a introduir codi dinàmic, no arriba a l'alçada de JSP.
- **Controlador:** és la capa on hi ha més diferències i on Django és molt més àgil i elegant. En Django, tots els controladors corresponen a funcions de Python mentre que en Spring cada controlador és una classe i existeixen diferents tipus de controladors segons el seu propòsit. Pel que fa al mapeig de URLs, en Django és molt més potent, elegant i no hi ha cap necessitat de passar paràmetres GET a les URLs.

Com a resum final, es podria dir que Spring és recomanable en el cas d'aplicacions web complexes i que necessiten l'escalabilitat i robustesa que pot donar un servidor d'aplicacions. En qualsevol altre cas i sobretot en petits projectes, Django és molt més recomanable ja que el desenvolupament és molt més ràpid i senzill. En el cas d'aquest treball, l'escalabilitat ens la dóna Google App Engine.

Bibliografia

[1] Python Documentation.
Disponible a http://www.python.org/doc/ (16/07/09)

[2] Django Documentation.
Disponible a http://docs.djangoproject.com/en/dev/ (16/07/09)

[3] Adrian Holovaty and Jacob Kaplan-Moss. "The Django Book".
Disponible també via web a http://www.djangobook.com/ (16/07/09)

[4] Google App Engine Documentation.
Disponible a http://code.google.com/intl/en/appengine/docs/ (16/07/09)

[5] AppEngineLearn. "App Engine Datastore".
Disponible a http://www.appenginelearn.com/ (16/07/09)

[6] Google App Engine Patch Wiki.
Disponible a http://code.google.com/p/app-engine-patch/ (16/07/09)

[7] Google Maps API.
Dispnible a http://code.google.com/intl/ca/apis/maps/ (16/07/09)

[8] Picasa Web Albums Data PAI.
Disponible a http://code.google.com/intl/ca/apis/picasaweb/ (16/07/09)

[9] jQuery Documentation.
Disponible a http://docs.jquery.com/Main_Page (16/07/09)

[10] Prototype API Documentation.
Disponible a http://www.prototypejs.org/api (16/07/09)

[11] Yahoo! UI Library Documentation.
Disponible a http://developer.yahoo.com/yui/editor/ (16/07/09)

[12] Dynarch Calendar Documentation.
Disponible a http://www.dynarch.com/projects/calendar/doc/ (16/07/09)

[13] Guido van Rossum. "Rapid Development with Python, Django, and Google App Engine". Google I/O 2008.

[14] Brett Slatkin. "Building Scalable Web Applications with Google App Engine". Google I/O 2008.

[15] Rafe Kaplan. "Working with Google App Engine Models". Google I/O 2008.

[16] Ryan Barrett . Ünder the Covers of the Google App Engine Datastore". Google I/O 2008.

[17] Brett Slatkin. "Building Scalable, Complex Apps on App Engine". Google I/O 2009.

Llicència